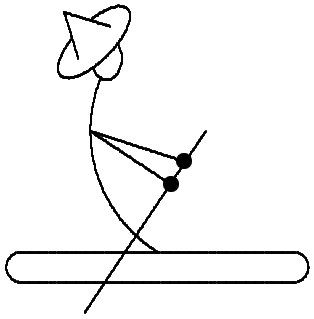


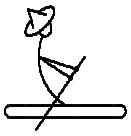
Charon User Manual and Reference Guide
Version 1.0

Rob F. Van der Wijngaart

August 24, 2000

Contents

Chapter 1



Introduction

This manual explains the functionality of the software library Charon. It also contains a number of actual programming examples in both Fortran 77 and C. Charon is intended for scientific programmers who are responsible for developing numerical software for the solution of partial differential equations on structured grids. This may entail either conversion of existing (vector) codes, or development from scratch.

Readers of this manual should be familiar with C or Fortran to understand the examples and declarations given, although the concepts on which Charon is based do not depend on the language of implementation. The reader is also assumed to have a basic grasp of the Message Passing Interface (MPI) standard [23], which specifies how processes on multi-computer systems exchange data, although only very few details are needed. Experience with message passing applications and parallel computing in general is an advantage, but it is not necessary. The benefit of such experience is that it makes it easier to appreciate why certain design choices were made.

1.1 How to read this manual

The subsequent chapters describe all the details of the current implementation of Charon, so that the experienced programmer can take full advantage of all its facilities. The novice parallel programmer may want to skip certain parts at first reading.
A recommended first introduction to the Charon way of developing parallel programs starts with Chapter 2, which presents a brief summary of the whole library.
Skim over Chapter 3, Sections 3.1 through 3.4, to understand the principles of Charon's data distribution, skipping any ugly details. But study the programming examples given in Section 3.5, looking up any functions used in the examples that were skipped earlier.
Read Chapter 4 to learn about how operations can be performed on distributed data sets, but skip Sections 4.2, 4.3 and 4.4. Of the distributed execution examples in Section 4.5, read only the first (Section 4.5.1).

All of the short Chapter 5 should be read, as it explains how a distributed program is turned into a true parallel program.
Chapter 6, among the most important of this manual, describes how Charon's grid data communication functions work. In a first reading, skip Section 6.2 on unstructured communications, and the corresponding example in Section 6.4.2.
Any programmer interested in reading and writing distributed arrays should read the short Chapter 7 on distributed I/O.
Although it is not important for the understanding of Charon, it may be useful to consult Chapter 8 on diagnostics once the reader starts writing and debugging programs.
Chapter 9 on advanced topics can probably be skipped altogether initially, unless the programmer wants to focus on converting vector codes to parallel codes.
Chapter 10 contains practical parallel programming advice and presents a number of examples the reader may want to examine before starting to use the library.

1.2 About the examples

The programming examples presented throughout the manual include simple techniques for simple problems, such as parallelization methods for explicit numerical algorithms. But they also explain more advanced techniques, such as pipelining, transpose-based methods, and multi-partitioning. The examples show how to accommodate legacy code constructions such as overindexing (for vector codes) and index reduction by defining low-dimensional sub-arrays of higher-dimensional arrays. Practical advice on how to implement staggered grids, where not all physical quantities are defined at coincident loci of the grid, is given as well. Cell-centered finite-volume grids are examples of staggered grids. We also dwell on how to accommodate multiple topologically independent grids.

1.3 Parallelization approaches

Many strategies have been devised in the past to facilitate writing programs for multi-processor computers. These roughly fall into the following categories:

- parallel languages and language extensions (HPC++ [14], HPF [15], Split C [8], etc.),
- parallelizing compilers and compiler directives for (virtual) shared-memory systems (SUIF [12], X3H5 [21], OpenMP [9], etc.),
- message passing (MPI [23], PVM [11], Linda [7], etc.).

Projects in each of these categories have enjoyed a certain level of success, but not a single winner has come out of all the academic and commercial parallelization projects [27]. Indeed, it can be argued that a completely general parallelization technique may never surface. At the same time, practical problems need to be solved. The logical approach is to narrow the focus and develop parallelization tools that are efficient and powerful within a certain application area.

1.4 Charon library built on message passing

Charon is a toolkit that enables the quick development of complicated scientific applications that are based on structured discretization grids. Such grids are common in the solution of partial differential equations that describe the physics of fluid flow, heat transfer, radiation, electron transport, etc. The toolkit takes the form of a set of library functions that can be called from C and Fortran programs. Charon is based on the message passing concept, and augments the *de facto* message passing standard MPI (Message Passing Interface). It can be installed and run on any system that supports MPI. It also works on single-processor non-MPI systems, in which case a partial MPI stub library (provided in the Charon package) must be linked.

No (pre-)compiler is provided. All functions and subroutines are implemented as (mostly standard) C and Fortran language constructs. The reason for this is to enhance portability of the library. It also frees us from the requirement of having to supply language parsers, which have inherent limitations.

1.5 Encapsulation in three-tiered design

The message passing approach to parallel scientific computing usually involves the distribution of data (arrays) among different processors. While this programming model is simple, it is also cumbersome. The programmer is responsible for the explicit restriction of data structures and operations to individual processors[1]. Certain packages, for example KeLP [10], OVERTURE [5], and PETSc [3], partly avoid this difficulty by providing encapsulation. That is, high-level distributed data structures are defined with minimal user input. Subsequently, the user can apply predefined operations to the distributed data. Mostly, this comes down to the specification of data parallel operations. This is very convenient when the numerical algorithm can be specified in data parallel terms, but it has two major disadvantages.

First, the user has little control over how the data is laid out in memory on the individual processor. Layout can greatly affect performance on modern computer systems with hierarchical memories. Second, this style of programming requires a definite change in program structure when a serial legacy code is to be converted. Moreover, if a certain operation is not present in the set of predefined operations, the user must somehow get access to the data and apply the operation 'by hand'. This sometimes requires copying data to user buffers, which is expensive, or involves the awkward mechanism of returning offsets from certain user-specified 'anchors' in memory (PETSc [4]).

In addition to functions that distribute the data, Charon provides parallelization tools at three levels of abstraction. At the top level, which is easiest to use, assignments to distributed array elements are completely encapsulated straightforward translations of assignments in a serial program or design. No change in program structure is required. As the programmer replaces high-level Charon constructs with mid-level functions, less encapsulation takes place, until, at the lowest level of abstraction, there is no encapsulation at all. Hence, a program being parallelized using Charon gradually changes from high to low levels of encapsulation. Since Charon functions

[1] We speak mostly of *processors* in this manual, although formally the word *processes* would usually be more appropriate.

at all levels of abstraction can be freely mixed, the programmer is never deprived of direct access to the data.

1.6 Language issues

Ease of use is one of the foremost goals of Charon. This sometimes results in unorthodox use of the languages in which the library has been implemented, most notably in Fortran 77. The ISO C implementation strays less from the standard. When Charon procedures are introduced, both the C and the Fortran syntax is presented. To make the distinction, we print all Fortran subprogram names in capitals, but the case sensitive C functions in the correct mixed-case format, with the first word following the `CHN` prefix (the *head*) capitalized. If the head is a verb, for example in `CHN_Assign`, the procedure is usually a subroutine (in Fortran), or a function returning an error code (in C). Such procedures are invoked for their side effects. If the head is a noun or adjective, for example in `CHN_Address`, the procedure is a function proper, whose prime result is its return value. We also indicate, in the style of the MPI Reference Manual [23], which of the arguments to Charon procedures are only read (`IN`), which are only written (`OUT`), and which are both written and read (`INOUT`). We follow the practice of marking a handle to a Charon variable as `OUT` or `INOUT` if the contents of the variable is changed, even if the handle itself is not affected.

1.6.1 Variable-length parameter lists

To keep the library small, Charon defines all operations independent of the dimensionality of the problem (number of spatial dimensions of the discretization grid). Some of these operations require as input a set of parameters of the length of the problem dimensionality, for example to specify the coordinates of a point in the grid. This can in principle always be handled by defining and filling arrays of the appropriate length and passing the starting address of the array to the function in question. However, this is very inconvenient, since it requires multiple assignmenmts by the user before every such function call.

To avoid this problem we use the mechanism of variable-length argument lists. This is adequately supported in C and Fortran 90, but not in Fortran 77. Rather than making an exception for old Fortran compilers, we adopt the common practice of defining the Fortran 77 functions with the longest expected number of parameters, but calling them with the actual number of parameters. Most current compilers will accept this practice, although some will issue an error if two function calls with different numbers of arguments occur in the same file. Often this situation can be avoided by placing the offending calls in different files. A future Fortran 90 implementation will fix this problem.

1.6.2 Function returning address

The Charon function `CHN_Address` returns a value that is used as an *lvalue* (address) by the `CHN_Assign` or `CHN_Invoke` routines. This is no problem in C, but Fortran does not allow it. Consequently, in Fortran such values are converted to ADDRESSTYPEs—often INTEGERs—that are the same size as C void pointers. The same holds for the return values of the functions `CHN_Mvalue` and `CHN_Start_address`. Determination of the actual Fortran type of ADDRESSTYPE

occurs during installation of Charon and is of no concern to the application programmer; it only reflects on the declaration of the return types of the three aforementioned functions in the include file `charonf.h`.

1.6.3 Choice variables

One of the most important Charon functions, `CHN_Create_distribution`, is used to create a distributed array, and to assign an amount of user-allocated memory to it; the function is passed the starting address of that memory. Distributions can have one of nine data types (five in Fortran, and four in C), and hence starting addresses can be one of nine types as well. To avoid having to define multiple instances of the same function for different data types, we regard the starting address as a choice variable that is cast to the right type inside `CHN_Create_distribution` (or `CHN_Set_start_address`).

1.6.4 Interoperability

Language interoperability refers to the possibility of passing data structures created in one language to program segments written in another language. For Charon the two choices for the language are C and Fortran. All Charon parameters and data structures are represented by integers (handles; see Section 1.8), which have the same meaning—and the same numerical value—in both C and Fortran. In MPI, however, data structures in C, though opaque to the user, need not be integers, and when Charon uses these, it matters whether they originate from a C or a Fortran program segment. Since MPI 1 does not provide language interoperability (see Section 1.12), Charon cannot either. That does not mean that all Charon programs must be written either completely in C or completely in Fortran. The three MPI constructs appearing in the Charon user interface are the communicator (`MPI_Comm`), the (elementary) data type (`MPI_Datatype`), and the reduction operator (`MPI_Op`). The only requirement that Charon poses is that these constructs be compatible, i.e. originating from the same language, whenever they are referenced together indirectly. For example, a grid variable defined in Fortran (and, hence, with a Fortran MPI communicator) cannot be used to define a distribution in C, with a C MPI data type.

1.7 Indexing

Many variables in Charon require indexing. Since the whole library is implemented in C, it is most natural to have all indices start at 0. Hence, cuts, coordinate dimensions, cells, etc. are all numbered starting with 0. The only exceptions are (possibly) the points in a grid (see Section 3.1) and the indices of a tensor (see Section 3). They are ordinarily also numbered starting with zero, but in order to be fully compatible with legacy code practices, these defaults can be overridden.

We reserve the word *indices* proper for grid points indices and tensor indices. *Subscripts* are used for indexing distributed arrays. The latter usually consist of a combination of grid point indices and tensor indices.

1.8 Handles

All Charon data structures created by the user are so-called opaque objects. They are represented by handles (aliases) that are of integer type in both C and Fortran. The contents of the data structures can not be accessed directly, but can only be changed by invoking additional Charon functions.

1.9 Error handling

When a Charon function is invoked with the wrong parameters or otherwise produces an error condition, the type of error returned depends on the function in question. All functions that are invoked strictly for their side effects (subroutines in Fortran) return an integer error code. Successful completion of such a function is indicated by the error code `CHN_SUCCESS`, which has the numerical value of zero. All other error codes are large negative numbers.

If the error is due to a faulty input value, it is termed a user error. See the list of error codes on page 117. Other errors are termed system errors. They are usually caused by insufficient memory, or by failing calls to the MPI library.

C functions return the error code as the return value, which can be ignored by the programmer. In the corresponding Fortran subroutine the error code is the last parameter in the parameter list. The reason for this asymmetry is that Fortran cannot ignore return values.

If a function returns an actual value and an error occurs, the result is a value that will usually not occur in user code. The particular error return values will be given in the sections that describe the functions.

1.10 Charon function execution modes

MPI defines five important distinguishing attributes of functions. In the current version of Charon, which does not support asynchronous communications at the user level, three of these types are relevant. For completeness, we paraphrase the MPI explanations of these types.

A procedure is *local* if its completion depends only on the local executing process. Such an operation does not require an explicit communication with another user process. Charon calls that generate local objects or that query the state of local objects are local (a local object is a Charon data structure created by and within the calling process).

A procedure is *non-local* if it may require the execution of some Charon procedure on another process. All Charon communications are non-local.

All Charon data structures have meaning only within a specific set of processes, called their process group (this group corresponds to an MPI communicator). A procedure is *collective* if all processes in the process group determined by the Charon data structures named in the procedure call must invoke the procedure, and with identical user arguments. An important variation is the set of *pseudo-collective* procedures. A procedure of this kind will always complete successfully (provided its arguments are valid and system memory suffices) if called as a collective procedure. But it only needs to be called by those processors whose state changes as a result of the execution of the procedure. For example, Charon communication routines can safely be skipped by those processors that neither send nor receive data.

We note that a procedure may be collective, but local. For example, the function creating a grid data structure must be passed the same dimensions and communicator on all the processes within that communicator, but no communication is necessary.

The execution mode of some Charon functions can be changed by another Charon function. For example, the function `CHN_REAL_VALUE`, whose mode is ordinarily non-local, can be changed to local through the function `CHN_Begin_local`.

1.11 Using Charon library

Unlike MPI, Charon does not require that an initialization routine be called. The user only need include the header file `charon.h` in C programs, and `charonf.h` in Fortran programs[2]. But if Charon is to be used in conjunction with MPI, the standard MPI initalization routine needs to be called before any calls to Charon and MPI are made [23, page 291]. Likewise, for proper termination of the MPI application, `MPI_Finalize` must be called at the end of the program. To obtain the proper MPI function prototypes (in C) and function return types, as well as the proper values for MPI constants, the standard MPI header files (`mpi.h` in C, `mpif.h` in Fortran) must be included *before* the Charon header file.

1.12 About the current version

Several features are planned for future release, but are not included in the current version of Charon. They are the following.

- Parallel I/O. With the specification of parallel I/O in the follow-up to the popular MPI standard, MPI 2 [18], it is now possible, in principle, to construct portable functions that utilize parallel file systems to read and write distributed variables or parts of them. A prototype implementation of parallel I/O functions for grid-based applications with good performance results was reported in [26]. However, because of the limited number of sites that have installed MPI 2 at present, it was decided not to rely on MPI 2 availability for this Charon release. Reading and writing distributed arrays from and to files is possible (see Chapter 7), but is limited to serial execution.

- Nonblocking, asynchronous communications. Although many of the Charon communications functions use nonblocking, asynchronous MPI communication primitives internally, the Charon functions themselves have blocking, synchronous semantics. In the next release of Charon, asynchronous versions of all communications will be provided.

- Message aggregation. Although some optimization of complicated Charon communications is already available through communication caching (see Section 9.3), the potentially more rewarding optimization of automatic message aggregation has not yet been implemented. This can be especially beneficial if many unstructured communications are used (`CHN_Get/Put_tile`, `CHN_Reduce/Bcast_tile`), or if the domain decompositions

[2] If the Fortran compiler does not support the *include* facility, the programmer must insert the text of the header file itself in the program.

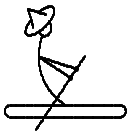
predefined within Charon are not sufficient. However, unless the majority of the Charon operations involve custom decompositions and unstructured communications, message aggregation will not affect performance much. Full implementation of message aggregation is planned for the next Charon release.

- *In situ* REDISTRIBUTION. Mapping one data distribution to another `CHN_Redistribute` currently requires the storage locations of source and target distributions to be completely disjoint. This allows relatively simple and swift copying of blocks of the distributed data, but may pose a memory problem. In the next Charon release the user will be allowed to specify (partially) overlapping memory locations, although it is recommended not to use this option unless absolutely necessary.

- FORTRAN LIMITS. Certain Charon functions accept a number of arguments that varies with the number of spatial dimensions of a grid (`CHN_Cell_index` and `CHN_Point_owner`), with the tensor rank of a distribution (`CHN_Create_distribution`), with the total number of subscripts of a distribution (`CHN_Address` and `CHN_Value` (several types)), with the number of fixed subscripts of a distribution (`CHN_Set_fixed_subscripts`), and with the number of buffers for a user-specified procedure (`CHN_Invoke`). In C there is no limit to this variable number of arguments, but in Fortran the maximum number for each of the spatial dimensions, subscripts, and buffers in the current version of Charon is nineteen.

- USER-DEFINED MPI DATA STRUCTURES. Charon currently circumvents the difficulty of MPI language interoperability by prohibiting the use of user-defined MPI data structures. The only impact this has on the library is that the programmer cannot customize communicators. Work is in progress that will remove this restriction, so that the limited language interoperability described in Section 1.6.4 will be supported. Once MPI 2 is widely available, full C/Fortran interoperability will be supported.

Chapter 2



Overview

Charon supports the parallelization of programs using multi-dimensional arrays related to structured grids. How this is done is explained in this chapter, which gives an overview of the whole library. Charon parallelization is based on the premise that the programmer knows best. Even fairly advanced parallel algorithms for structured-grid applications can usually be explained quite easily, which means that most algorithms are conceptually simple. It is the implementation that is frequently complex. Charon makes that complexity manageable by offering certain 'bookkeeping' services to the programmer, and by providing a vehicle for piecemeal transition from a serial to a parallel program.

2.1 Domain decomposition and data distribution

Most algorithms of our interest that are meant for distributed-memory parallel computers are based on domain decompositions; the amount of computational work per point in a grid is often fixed, so by distributing equal-sized sets of points to different processors, all have the same amount of work to do. With every point is usually associated a fixed amount of data. For example, in a three-dimensional compressible fluid flow calculation one will want to store at each grid point the values of density, x-, y-, and z-momentum, and the specific energy. Depending on the complexity of the algorithm, more data elements will be stored at each grid point. If the same processor keeps working on the same set of points throughout the calculation, most of the data associated with these points can be kept local (in that processor's memory).

Often stencil operations are part of the algorithm, which means that for computation of values at grid points, values from neighboring grid points are needed. If these points belong to other processors, retrieving the requested values may take a long time, due to the slowness of so-called interprocessor communication. Hence, it is beneficial to keep both the number and size of such requests to a minimum. This is accomplished at least in part by distributing geometrically compact subsets of the grid points to the different processors.

Compact sets have a small surface-to-volume ratio. Whereas the amount of useful compu-

tational work is proportional to the volume of the set, the amount of communication overhead is proportional to the surface area. In physical space the (hyper)sphere has the most favorable surface-to-volume ratio, as is evidenced by the result of the action of surface tension on isolated drops of mercury. In computational space, the best surface-to-volume ratios are obtained by Cartesian-product subsets of points with unit aspect ratios (squares in two dimensions, cubes in three dimensions, etc.). Consequently, the best domain decomposition strategy for structured-grid applications would appear to be that which divides each grid into cubical blocks or cells of equal size, and assigns each cube to a different processor. This is true for algorithms without strong data dependencies (see below), but not for many others of interest to the scientific computing community.

If the numerical algorithm allows that the points in the grid be visited in any order, for example in a point-Jacobi update scheme, the algorithm is said to be naturally data parallel. Another way of expressing this property is by stating that the numbering of points does not influence the result of applying the algorithm. If the order in which points are visited does matter, for example in a Gauss-Seidel update scheme, then data dependencies are said to exist, and the algorithm may no longer be data parallel. Renumbering points changes the outcome of the method.

Strong data dependencies influence the optimal way in which a grid should be distributed (partitioned), and the particular partitioning scheme is best specified by the programmer. The data distribution process consists of four fundamental steps:

1. Define a *grid* data structure that specifies the dimensionality of the problem, and the number and indices of the points at which array variables may be defined.

2. Create a partitioning in *cells* (grid sub-blocks), which are created by intersecting the grid with cutting planes (*cuts*) that are parallel to coordinate planes. The result is a *section* data structure.

3. Assign cells to processors. The result is a *decomposition* data structure.

4. Create the multi-dimensional, distributed array and associate it with a decomposition and local storage space. The result is a *distribution* data structure.

The reason why we divide the distribution process into four steps, rather than collapsing it into a single one, is that this maximizes user control and provides an easy means of testing parallel algorithms. At the same time, the four data structures have a natural meaning that should be easy to master by the programmer. Each data structure can serve as the basis for defining many different derived (higher-numbered) data structures.

For example, a user may divide a grid into ten slices, with the intent of solving a problem on as many processors. Hence, the section data structure contains nine cuts. But the code may be tested on a single processor by assigning all cells (slices) to the same processor when defining the decomposition. In turn, a single decomposition of the grid can be used to distribute scalar, vector, or tensor fields. When the abovementioned ten-slice uni-processor code is ultimately run on the intended ten processors, all the user needs to change is the definition of the decomposition, keeping all the grid, section, and distribution variables the same.

2.2 Distributed and concurrent execution

Once distributed variables have been defined for all the grid arrays in the application, the assignments involving elements of the arrays have to be cast in a form that Charon can interpret. They also have to be scheduled so that data dependencies are respected, while making sure that all processors have work to do at all times. Developing such a schedule so that a balanced load and truly concurrent execution ensues, is nontrivial for many applications. It is the source of many coding errors, because the data distribution has an impact on virtually all program statements. It would be easier if the program could first be written without concern for the distribution—generally very inefficient—and be tuned for performance later.

This is the approach taken by Charon. First, all assignments to elements of arrays in the serial code (or code design) are replaced by library calls (`CHN_Assign`). These take into account the distributed variables on left and right hand side, and the points in the grid at which these variables ought to be evaluated (right hand side: `CHN_Value`) or assigned (left hand side: `CHN_Address`). Other than that, all program logic remains serial. Charon inspects (the components of) the right hand side, determines which processors contribute to it, and arranges for them to furnish the processor responsible for updating the left hand side with the concerned values. This happens behind the scenes, and the user does not have worry about any details of the data distribution or communication requests while inserting `CHN_Assign` calls. All remote data requests are implicitly invoked and satisfied.

All processors execute the same (serial) code, so no speed gain is obtained. Moreover, since each assignment carries with it the possibility of a communication, very many data exchanges between processors will occur, which slows the program execution even more. But the important accomplishment is that the single processor code has been moved to a distributed-memory platform in a simple and foolproof fashion that guarantees that an originally correct implementation remain correct.

Subsequently, the user inserts calls that instruct Charon when it is safe for certain processors to skip some of the statements and when to suppress requests for data from remote processors (`CHN_Begin/End_local`). In the case of stencil operations, assignments to points at the boundaries of cells owned by the calling processor still require data from geometrically adjacent processors. Such requirements must now be satisfied explicitly by the programmer, namely by copying data from the 'faces' of adjacent cells to the calling processor. This is done by the function `CHN_Copy_faces`, which copies data from adjacent cells into a border of auxiliary or 'ghost' points on the calling processor.

Another way of explicitly satisfying remote data requests may be obtained by redistributing the grid (`CHN_Redistribute`). For example, an application programmer may originally have partitioned a grid in the x-coordinate direction. But when a program segment is entered in which frequent reference is made to points with offsets only in the x-direction and not in the y-direction, the programmer may switch to a partitioning that divides the y-direction but keeps the x-direction undivided. This is generally called a transposition operation. Other kinds of redistributions are also available in Charon.

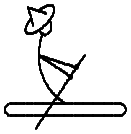
In case of non-stencil operations, the `CHN_Copy_faces` and `CHN_Redistribute` routines may not suffice to retrieve data from remote processors. The function `CHN_Get_tile` can be used to fetch arbitrary Cartesian-product subsets of the distributed array and store them on the calling processor.

The process of bypassing implicitly invoked communications and restricting execution of certain program fragments to certain processors is error prone. But because we start from a correct distributed implementation, we need only convert small parts of the code at a time, and can leave the rest unchanged. When the whole code has been converted, the program is both distributed and concurrent.

Finally, the expensive library calls `CHN_Assign`, `CHN_Address`, and `CHN_Value` must be eliminated altogether. At this point, the programmer must use local indexing for those parts of the distributed arrays owned by the calling processor. This process also potentially leads to errors, but again it can be undertaken one step at a time. Moreover, query functions in Charon help the programmer in keeping the index bookkeeping straight.

Many parallel single-grid applications can be coded efficiently using only Charon communication and query functions. However, certain important multi-zone or multi-discipline applications cannot be fully served by Charon. For this purpose the MPI library is available to the user, who can always arrange for inter-grid or inter-discipline communications to occur through plain message passing. MPI may also be used if the user is not satisfied with the communication performance or functionality of Charon. This places another level of performance tuning at the disposal of the programmer.

Chapter 3



Data distribution support tools

3.1 Grids

The programmer wishing to define a discretization grid of a certain dimensionality calls the function `CHN_Create_grid`. It allocates space for the variable and sets the handle `grid` to a positive value (zero in case of failure). One of the inputs is an MPI communicator created by the user, or the predefined `MPI_COMM_WORLD`, which contains all the processors allocated to the parallel program. The communicator is one of the few direct links between Charon and MPI visible to the programmer (the others are `MPI_Datatype`, used in the definition of distributions (Section 3.4) and `MPI_Op` used in the reduction operation `CHN_Reduce_tile` (page 74)). Its significance is that in operations on variables derived from the grid that require cooperation of processors, at most the processors in the communicator will participate. If the calling processor is not a member of the communicator, the creation completes successfully, but an invalid grid handle is returned.

All functions that create, modify, or destroy a grid must be called by all the processors in the communicator, with the same parameters; they are local, collective operations.

```
int CHN_Create_grid(int *grid, MPI_Comm comm, int num_dims)
CHN_CREATE_GRID(grid, comm, num_dims, ierr)
   integer grid, comm, num_dims, ierr
```

OUT	`grid`	handle to grid data structure
IN	`comm`	handle to MPI communicator
IN	`num_dims`	number of spatial dimensions

CAUTION: The communicator used in the definition of the grid data structure is used by Charon communication functions, and it is possible that these suffer from interference caused by (asynchronous) MPI user communications employing the same communicator. This can be avoided by using a duplicate communicator (`MPI_Comm_dup`) in the grid definition.

Setting the size of the grid in a particular coordinate direction is done using CHN_Set_grid_size. Only strictly positive grid sizes are allowed. It is important to recognize that no actual coordinate values at the grid points are specified by this function, but only the range of coordinate *indices*. Coordinate values of the generally curvilinear grid can be stored in a distributed variable (see Section 3.4). Once its sizes in all coordinate dimensions are defined, the grid is ready for use.

```
int CHN_Set_grid_size(int grid, int dir, int size)
CHN_SET_GRID_SIZE(grid, dir, size, ierr)
   integer grid, dir, size, ierr
```

INOUT	grid	handle to grid data structure
IN	dir	coordinate direction
IN	size	dimension of grid

By default grid indices start with 0, so setting a grid size to n makes available indices 0 through $n-1$. The default can be overridden by the function CHN_Set_grid_start_index. Any integer starting index (possibly negative) is allowed.

```
int CHN_Set_grid_start_index(int grid, int dir, int start_index)
CHN_SET_GRID_START_INDEX(grid, dir, start_index, ierr)
   integer grid, dir, start_index, ierr
```

INOUT	grid	handle to grid data structure
IN	dir	coordinate direction
IN	start_index	starting index of grid

Finally, a grid variable can be destroyed, and its handle reset to zero, by the CHN_Delete_grid function. A grid should not be deleted or modified if other Charon variables derived from it are still in use.

```
int CHN_Delete_grid(int *grid)
CHN_DELETE_GRID(grid, ierrr)
   integer grid, ierr
```

INOUT	grid	handle to grid data structure

3.1.1 Grid query functions

Query functions return the parameters that make up the grid. These are the already defined dimensionality, communicator, sizes, and starting indices. For convenience we also provide the ending indices of the grid, and the rank of the calling processor within the communicator.

```
int CHN_Grid_dimensionality(int grid)
integer function CHN_GRID_DIMENSIONALITY(grid)
   integer grid
```
Error return value: Greatest representable negative integer

IN	grid	handle to grid data structure

```
MPI_Comm CHN_Grid_comm(int grid)
integer function CHN_GRID_COMM(grid)
   integer grid
```
Error return value: `MPI_COMM_NULL`

IN	grid	handle to grid data structure

```
int CHN_Grid_size(int grid, int dir)
integer function CHN_GRID_SIZE(grid, dir)
   integer grid, dir
```
Error return value: Greatest representable negative integer

IN	grid	handle to grid data structure
IN	dir	coordinate direction

```
int CHN_Grid_start_index(int grid, int dir)
integer function CHN_GRID_START_INDEX(grid, dir)
   integer grid, dir
```
Error return value: Greatest representable positive integer

IN	grid	handle to grid data structure
IN	dir	coordinate direction

```
int CHN_Grid_end_index(int grid, int dir)
integer function CHN_GRID_END_INDEX(grid, dir)
   integer grid, dir
```
Error return value: Greatest representable negative integer

IN	grid	handle to grid data structure
IN	dir	coordinate direction

```
int CHN_Rank(int grid)
integer function CHN_RANK(grid)
   integer grid
```
Error return value: -1

IN	grid	handle to grid data structure

3.2 Sections

A section data structure is used to indicate how a discretization grid is divided into a number of contiguous blocks of grid points (cells). It is based on a previously created, completed Charon grid data structure. After the section is initialized, cuts are made, meaning cutting planes parallel to coordinate planes are defined. This can be done one by one, or by calling one of the predefined cutting strategies used for creating common domain decompositions. Cells, demarcated by cutting planes and/or the boundaries of the grid, are Cartesian-product subspaces. Together they form a disjoint covering of the grid, so a loop over all the grid points in all the cells visits each point in the grid exactly once. The word 'section' is short for Cartesian section. All functions that create, modify, or destroy a section, are local and collective. Query functions are local.

```
int CHN_Create_section(int *section, int grid)
CHN_CREATE_SECTION(section, grid, ierr)
   integer section, grid, ierr
```

OUT	section	handle to section data structure
IN	grid	handle to grid data structure

Upon creation a section is automatically initialized with -1 cuts—an invalid number—in all coordinate directions. Cuts can be placed using elementary Charon operations. First, the programmer must declare the number of cuts in the relevant coordinate direction.

```
int CHN_Set_num_cuts(int section, int dir, int num_cuts)
CHN_SET_NUM_CUTS(section, dir, num_cuts, ierr)
   integer section, dir, num_cuts, ierr
```

INOUT	section	handle to section data structure
IN	dir	coordinate direction
IN	num_cuts	number of cuts in this coordinate direction

Actual cuts are either placed individually, or all at once, for a certain coordinate direction. A cut value of n means that a cutting plane is inserted between grid point indices $n-1$ and n. Cuts must be placed in strictly increasing order of grid point index, that is, a cut with sequence number k has a cut value that is strictly smaller than a cut with sequence number $k+1$. Cut sequence numbers depend on the relative locations of the cuts within the relevant coordinate direction of the grid, not on the order in which they are created.

```
int CHN_Set_cut(int section, int dir, int cut_num, int cut_value)
CHN_SET_CUT(section, dir, cut_num, cut_value, ierr)
   integer section, dir, cut_num, cut_value, ierr
```

INOUT	section	handle to section data structure
IN	dir	coordinate direction
IN	cut_num	sequence number of cut
IN	cut_value	point index *after* cut

Figure 3.1 shows a legal cutting of a two-dimensional grid. Point indices start with two in the first coordinate direction, and with zero (the default) in the second. Notice that the first coordinate direction is always numbered zero (both C and Fortran), and so is the first cut in any particular coordinate direction.

While `CHN_Set_cut` is very flexible, it is cumbersome for many ordinary domain decompositions. Often a programmer will simply want to divide a grid in a certain direction as evenly as possible, given a certain number of cuts. Alternatively, the programmer may want to specify a certain spacing s, meaning that cuts should be placed exactly s points apart. The following two functions accomplish this.

```
int CHN_Set_even_cuts(int section, int dir)
CHN_SET_EVEN_CUTS(section, dir, ierr)
   integer section, dir, ierr
```

INOUT	section	handle to section data structure

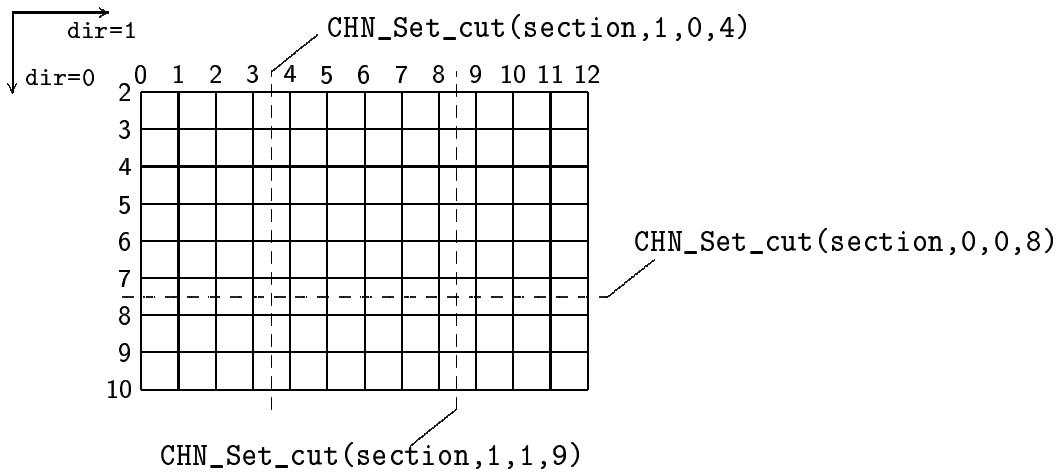


Figure 3.1: Completed two-dimensional section with three cuts

IN	dir	coordinate direction

```
int CHN_Set_group_size(int section, int dir, int group_size)
CHN_SET_GROUP_SIZE(section, dir, group_size, ierr)
   integer section, dir, group_size, ierr
```

INOUT	section	handle to section data structure
IN	dir	coordinate direction
IN	group_size	constant spacing between cuts

Usually the specified number of cuts does not divide the grid exactly evenly. Assume n_c cuts and a grid of size n_g. Calling `CHN_Set_even_cuts` results in the following. The first $n_g \bmod (n_c+1)$ cuts will be spaced $\lfloor n_g/(n_c+1) \rfloor + 1$ points from the predecing cut (or, for the first cut, from the grid boundary). The remaining cuts are spaced $\lfloor n_g/(n_c+1) \rfloor$ points apart. Of course, the number of cuts in the particular coordinate direction has to be specified first. This is not required by `CHN_Set_group_size`, which computes the number of cuts from the constant inter-cut spacing. When the group size does not divide the grid exactly evenly, only the cell between the last cut and the 'end' of the grid will be shortened.

A section data structure is deleted, and its handle reset to zero, by the `CHN_Delete_section` function. A section should not be deleted or modified if other Charon variables derived from it are still in use.

```
int CHN_Delete_section(int *section)
CHN_DELETE_GRID(section, ierr)
   integer section, ierr
```

INOUT	section	handle to section data structure

3.2.1 Predefined sections

Sections for the vast majority of applications can be created even more easily using one of three predefined sectioning routines. For example, almost all practical methods published over the last

decade for the solution of discretized partial differential equations on structured grids assign to each processor one contiguous grid block, or cell. Such a domain decomposition is called a *uni-partition*. Given the number of processors P that are members of the communicator on which the grid is based, it is often possible to compute a sectioning that has exactly that many roughly equal-sized cells. Exceptions may occur, depending on the uni-partioning strategy, when P or its integral factors are large compared to the grid size.

For instance, a user may insist on the uni-partitioning of a three-dimensional grid of $17 \times 87 \times 73$ points over 137 processors. But 137 is prime and exceeds the size of the grid in all coordinate directions. Hence, no possible cutting of the grid can result in the required number of cells. If the number of processors were $140 (= 2*(2*5)*7)$ instead, then placing 1, 9, and 6 evenly spaced cuts in coordinate directions 0, 1, and 2, respectively, would produce the desired result. These cuts are chosen such that the created cells are as close to cubical as possible, which minimizes their aspect ratios and hence their (communication) surface area. Another possibility is to choose the number of cuts as even as possible in all coordinate directions (in this case 3, 4 and 6 cuts, respectively), which minimizes the total number of cuts. These two different strategies are associated with values of the `shape` parameter of `CHN_DEFAULT_SHAPE` and `CHN_EQUAL_CUTS`, respectively. The shape parameter is all that is required to prepare a section suited for the uni-partition domain decomposition. Figure 3.2 illustrates its effect on the cutting of a two-dimensional grid for sixteen processors.

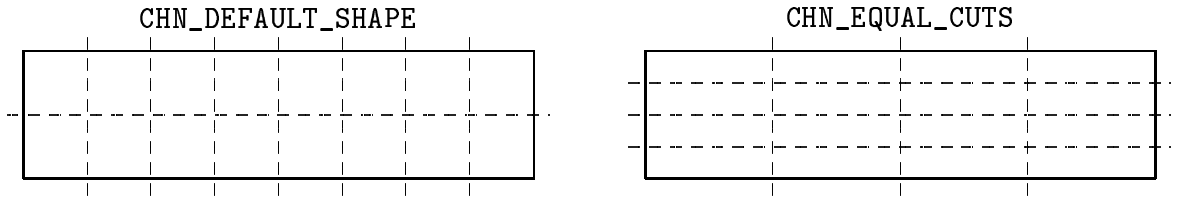


Figure 3.2: Effect of shape parameter on uni-partition section

```
int CHN_Set_unipartition_cuts(int section, int shape)
CHN_SET_UNIPARTITION_CUTS(section, shape, ierr)
   integer section, shape, ierr
```

INOUT	section	handle to section data structure
IN	shape	switch; `CHN_DEFAULT_SHAPE` or `CHN_EQUAL_CUTS`

Another useful predefined sectioning strategy prepares for the so-called *multi-partition* domain decomposition, which is explained in detail in Section 3.4. In this case every processor in the communicator receives not one but multiple cells, arranged in a regular pattern. In n-dimensional space a multi-partition decomposition is possible only if the number of processors P can be written as P_0^{n-1}, where P_0 is an integer (and if the grid size is at least P_0 in all dimensions). This poses no divisibility constraints in two dimensions, but requires that the number of processors be a square in three dimensions, a cube in four, etc. The resulting section contains the same number of cuts $(P_0 - 1)$ in all coordinate directions.

```
int CHN_Set_multipartition_cuts(int section)
CHN_SET_MULTIPARITION_CUTS(section, ierr)
```

```
   integer section, ierr
```

INOUT section	handle to section data structure

When defining uni- and multi-partitions it is sometimes convenient to exclude one or more coordinate directions from partitioning. For example, a three-dimensional grid may be divided into slices (one-dimensional partitioning) because of strong data dependencies within each slice. In this case the programmer could place all the cuts 'by hand'. But it is also possible to inform Charon that certain coordinate directions should not be divided, which will be respected by the functions creating the predefined sections. In that case the remaining coordinate directions will be divided using the lower-dimensional partitioning algorithm.

```
int CHN_Exclude_partition_direction(int section, int dir)
CHN_EXCLUDE_PARTITION_DIRECTION(section, dir, ierr)
   integer section, dir, ierr
```

INOUT	section	handle to section data structure
IN	dir	coordinate direction to be excluded

Both `CHN_Set_unipartition_cuts` and `CHN_Set_multipartition_cuts` apply `CHN_Set_even_cuts` internally to those coordinate directions that need to be divided.

The last predefined section prepares for the trivial *solo-partition* domain decomposition. It consists of a single cell, assigned to one processor. This simple operation defines zero cuts in all coordinate directions.

```
int CHN_Set_solopartition_cuts(int section)
CHN_SET_SOLOPARITION_CUTS(section, ierr)
   integer section, ierr
```

INOUT section	handle to section data structure

Fine tuning of cuts placed automatically by `CHN_Set_even_cuts`, `CHN_Set_group_size`, and `CHN_Set_uni/multipartition_cuts` can be achieved by calling `CHN_Set_cut`, as long as the sequence number of the cut and the new cut value are valid. If the number of cuts in a certain coordinate direction must be changed, the programmer can call `CHN_Set_num_cuts`, but any information on previously defined cut values is lost.

3.2.2 Section query functions

Query functions are defined that return for a given section the handle to the grid data structure on which it is based, the number of cuts in a certain coordinate direction, and the value of a particular cut, respectively.

```
int CHN_Grid(int section)
integer function CHN_GRID(section)
   integer section
```

Error return value: -1

IN	section	handle to section data structure

```
int CHN_Num_cuts(int section, int dir)
integer function CHN_NUM_CUTS(section, dir)
   integer section, dir
```
Error return value: -1

IN	section	handle to section data structure
IN	dir	coordinate direction

```
int CHN_Cut(int section, int dir, int cut_num)
integer function CHN_CUT(section, dir, cut_num)
   integer section, dir, cut_num)
```
Error return value: Greatest representable negative integer

IN	section	handle to section data structure
IN	dir	coordinate direction
IN	cut_num	sequence number of cut

3.3 Decompositions

The Charon decomposition data structure is derived from the section data structure. It is used to store information on the ownership of individual cells defined by the section. All functions that create, manipulate or destroy decompositions are local and collective. Query functions are local.

The decomposition data structure is initialized by `CHN_Create_decomposition`.

```
int CHN_Create_decomposition(int *decomposition, int section)
CHN_CREATE_DECOMPOSITION(decomposition, section, ierr)
   integer decomposition, section, ierr
```

OUT	decomposition	handle to decomposition data structure
IN	section	handle to section data structure

Ownership of a cell is defined as the sequence number (MPI rank within the communicator) of the processor that is associated with that cell. With some exceptions, elements of distributed variables corresponding to a cell owned by processor p can only be changed by that processor. We note that several different decompositions can be derived from the same section. For example, we may call `CHN_Set_unipartition_cuts` to define a section whose cuts demarcate 18 disjoint cells if the communicator contains 18 processors. Each of these cells may subsequently be assigned to a different processor, but they may also all be assigned, for example, to the same processor. Cell ownership is defined by `CHN_Set_cell_owner`. It takes as an argument the sequence number of a cell. This sequence number can be obtained from the vector of indices of the cell within the decomposition, through a call to the query function `CHN_Cell_index` (see page 28).

```
int CHN_Set_cell_owner(int decomposition, int rank, int c)
CHN_SET_CELL_OWNER(decomposition, rank, c, ierr)
   integer decomposition, rank, c, ierr
```

INOUT	decomposition	handle to decomposition data structure
IN	rank	rank of processor in communicator
IN	c	global sequence number of cell within decomposition

A decomposition data structure is deleted, and its handle reset to zero, by the function `CHN_Delete_decomposition`. A decomposition should not be deleted or modified if other Charon variables derived from it are still in use.

```
int CHN_Delete_decomposition(int *decomposition)
CHN_DELETE_DECOMPOSITION(decomposition, ierr)
   integer decomposition, ierr
```

`INOUT decomposition`	handle to decomposition data structure

3.3.1 Predefined decompositions

Completely arbitrary decompositions can be defined using `CHN_Set_cell_owner`, but most useful decompositions are not arbitrary. Several special decompositions are predefined, namely the solo-, uni-, and multi-partitions. These are constructed as follows.

`CHN_Set_solopartition_owners` assigns all cells to the same root processor, designated by the programmer. Note that this function can take as input sections of any kind, not just those created by `CHN_Set_solopartition_cuts`.

`CH_set_unipartition_owners` assigns each cell to a different processor. Numbering is in lexicographical order of cell indices, meaning that in a section with c_0 cells in the first coordinate direction, the cell with indices $(i, 0, 0, \ldots, 0)$ is assigned to processor i, that with indices $(1, j, 0, \ldots, 0)$ to processor $c_0 + j$, etc. A cell index is determined by the cuts that demarcate it. Cell (i, j, k, l) lies between the cuts $i - 1$ and i, cuts $j - 1$ and j, cuts $k - 1$ and k, and cuts $l - 1$ and l in the first, second, third, and fourth coordinate direction, respectively. A cut with index -1 is understood to signify the lower boundary of the grid, although this is not actually part of the section. Similarly, cut c_0 in the example above is not actually part of the section, but corresponds to the upper boundary of the grid. The easiest way of composing a section that can serve as a basis for a uni-partition decomposition is through invocation of function `CHN_Set_unipartition_cuts`, although any section that defines as many cells as processors is appropriate.

`CHN_Set_multipartition_owners` assigns to every processor multiple cells in a regular pattern that has the following remarkable property. Each 'hyperslice' of cells (collection of cells between two cuts in n-dimensional space) in every coordinate direction contains exactly one cell of each processor. Such an arrangement is constructed fairly easily, provided the number of processors P can be written as P_0^{n-1}, where P_0 is an integer (see Section 3.2). The section must contain exactly $P_0 - 1$ cuts in all coordinate directions, which creates P_0 hyperslices in every coordinate direction, containing P_0^{n-1} cells each.

The easiest way of composing a section that can serve as a basis for a multi-partition decomposition is through invocation of function `CHN_Set_multipartition_cuts`, although any section that defines the correct pattern of cuts is appropriate. Each cell in the hyperslice whose *last* cell index is 0 is assigned to a different processor. Since there are P_0^{n-1} processors in the communicator, this means that each processor receives exactly one cell in that hyperslice. Assignment of ownership within the hyperslice follows the pattern of `CHN_Set_unipartition_owners`. Ownership of cell $(i_0, i_1, \ldots, i_{n-2}, i_{n-1})$—indicated here with function $\mathcal{O}$—in the hyperslice whose

last index is i_{n-1} ($i_{n-1} > 0$), relates to that in the first slice as follows:

$$\begin{aligned}\mathcal{O}(i_0, i_1, \ldots, i_{n-2}, i_{n-1}) = \mathcal{O}((&(i_0 + i_{n-1}) \bmod P_0,\\ &(i_1 + i_{n-1}) \bmod P_0,\\ &\ldots,\\ &(i_{n-2} + i_{n-1}) \bmod P_0,\\ &0)\,.\end{aligned}$$

In other words, after the cells in the first hyperslice have been assigned to processors, ownership within each subsequent slice is determined by (cyclically) shifting the ownership pattern of the preceding slice. A 3D multi-partition decomposition for 16 processors is depicted in Figure 3.3.

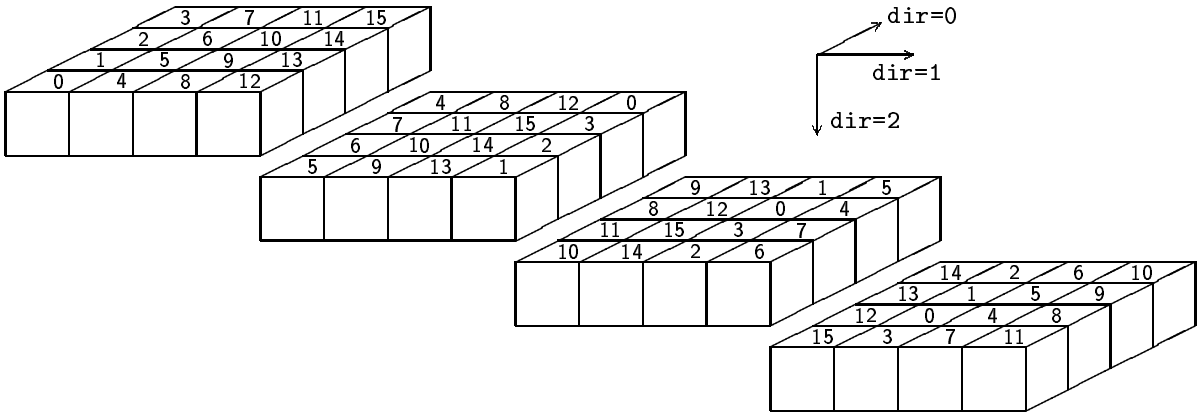
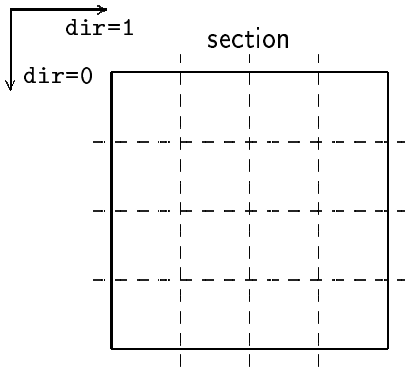


Figure 3.3: Three-dimensional multi-parition decomposition for 16 processors (exploded view); processor ownership is indicated within cells

There are many decompositions conceivable that share with the predefined Charon multi-partition the property that each hyperslice contain exactly one cell owned by each processor. The Charon decomposition is special in three respects. First, there is no decomposition with this property that has fewer cuts. Second, the same neighbor relation holds for all cells owned by a certain processor. For example, as can be seen in Figure 3.3, each cell owned by processor two is adjacent to one owned by processor six in the positive second coordinate direction. This property is important when transferring data in bulk among all cells in the decomposition (`copy_faces_all`; Section 6.1.1). Third, and least importantly, in a hypercube architecture the Charon multi-partition minimizes the maximum number of hops (network routers to be traversed) for nearest-neighbor communication [6].

Tip: If it is possible to use one of the predefined Charon decompositions, it is always advisable to do so. Not only is it more convenient and less error-prone than assigning processor ownership by hand, it is also more efficient. Predefined decompositions are marked by Charon, and their special properties are exploited to optimize communications.

Figure 3.4 shows an example of a single section leading to three different, predefined decompositions. For the solo-partition processor three has been designated by the user as the root processor.



CHN_Set_solopartition_owners

3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3

CHN_Set_unipartition_owners

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

CHN_Set_multipartition_owners

0	1	2	3
1	2	3	0
2	3	0	1
3	0	1	2

Figure 3.4: Section used to create three different decompositions

```
int CHN_Set_solopartition_owners(int decomposition, int root)
CHN_SET_SOLOPARTITION_OWNERS(decomposition, root, ierr)
   integer decomposition, root, ierr
```

INOUT	decomposition	handle to decomposition data structure
IN	root	rank of processor owning all cells

```
int CHN_Set_unipartition_owners(int decomposition)
CHN_SET_UNIPARTITION_OWNERS(decomposition, ierr)
   integer decomposition, ierr
```

INOUT	decomposition	handle to decomposition data structure

```
int CHN_Set_multipartition_owners(int decomposition)
CHN_SET_SOLOPARTITION_OWNERS(decomposition, ierr)
   integer decomposition, ierr
```

INOUT	decomposition	handle to decomposition data structure

3.3.2 Decomposition query functions

The functions CHN_Section, CHN_Num_cells, CHN_Total_num_cells and CHN_Total_num_owned_cells return a handle to the section on which the decomposition is based, the number of cells in the decomposition in a certain coordinate direction, the total number of cells in the

decomposition, and the total number of cells in the decomposition owned by the calling processor, respectively. If not all cells have been assigned proper ownership yet, some of the query functions will return invalid values.

```
int CHN_Section(int decomposition)
integer function CHN_SECTION(decomposition)
   integer decomposition
```
Error return value: -1

IN	`decomposition`	handle to decomposition data structure

```
int CHN_Num_cells(int decomposition, int dir)
integer function CHN_NUM_CELLS(decomposition, dir)
   integer decomposition
```
Error return value: -1

IN	`decomposition`	handle to decomposition data structure
IN	`dir`	coordinate direction

```
int CHN_Total_num_cells(int decomposition)
integer function CHN_TOTAL_NUM_CELLS(decomposition)
   integer decomposition
```
Error return value: -1

IN	`decomposition`	handle to decomposition data structure

```
int CHN_Total_num_owned_cells(int decomposition)
integer function CHN_TOTAL_NUM_OWNED_CELLS(decomposition)
   integer decomposition
```
Error return value: -1

IN	`decomposition`	handle to decomposition data structure

Many Charon functions take as an input the sequence number of a cell. This number can be obtained from the coordinates of the cell within the decomposition, using `CHN_Cell_index`. For example, the coordinates of the cell owned by processor eight in in the uni-partition decomposition in Figure 3.4 are: $(0, 2)$. Hence, the cell index is returned by: `CHN_Cell_index(decomposition, 0,2)`. Redundant additional coordinates are ignored by the query function. In ISO C the `stdarg` facility is used to implement the variable-length argument list. In Fortran an ad hoc solution is used that works with most modern compilers.

```
int CHN_Cell_index(int decomposition, ...)
integer function CHN_CELL_INDEX(int decomposition, index0, index1, ...)
   integer index0, index1, ...
```
Error return value: -1

IN	`decomposition`	handle to decomposition data structure
IN	`index0`	first cell coordinate
IN	`index1`	second cell coordinate
IN	`...`	subsequent cell coordinates

Usually, each processor owns only a fraction of the total number of cells in the decomposition. The P_{own} cells owned by a certain processor are numbered locally as 0 through $P_{own} - 1$. Translation between local (within processor) and global (within overall decomposition) cell sequence number is accomplished by the following two functions. If a local number is requested of a cell not owned by the calling processor, the value -1 is returned.

```
int CHN_Own_to_global_cell_index(int decomposition, int c)
integer function CHN_OWN_TO_GLOBAL_CELL_INDEX(decomposition, c)
   integer decomposition, c
```
Error return value: -1

IN	decomposition	handle to decomposition data structure
IN	c	local sequence number of cell within processor

```
int CHN_Global_to_own_cell_index(int decomposition, int c)
integer function CHN_GLOBAL_TO_OWN_CELL_INDEX(decomposition, c)
   integer decomposition, c
```
Error return value: -1

IN	decomposition	handle to decomposition data structure
IN	c	global sequence number of cell within decomposition

Once the global cell number is known, the user can ask to know which processor owns the cell, and, for a certain coordinate direction, what is the coordinate value of the cell within the decomposition, what is the size of the cell (number of grid points), what is its starting grid point index, and what its ending grid point index, respectively.

```
int CHN_Cell_owner(int decomposition, int c)
integer function CHN_CELL_OWNER(decomposition, c)
```
Error return value: -1

IN	decomposition	handle to decomposition data structure
IN	c	global sequence number of cell within decomposition

```
int CHN_Cell_coordinate(int decomposition, int dir, int c)
integer function CHN_CELL_COORDINATE(decomposition, dir, c)
   integer decomposition, dir, c
```
Error return value: -1

```
int CHN_Cell_size(int decomposition, int dir, int c)
integer function CHN_CELL_SIZE(decomposition, dir, c)
   integer decomposition, dir, c
```
Error return value: -1

```
int CHN_Cell_start_index(int decomposition, int dir, int c)
integer function CHN_CELL_START_INDEX(decomposition, dir, c)
   integer decomposition, dir, c
```
Error return value: Greatest representable positive integer

```
int CHN_Cell_end_index(int decomposition, int dir, int c)
integer function CHN_CELL_END_INDEX(decomposition, dir, c)
   integer decomposition, dir, c
```
Error return value: Greatest representable negative integer

IN	decomposition	handle to decomposition data structure
IN	dir	coordinate direction
IN	c	global sequence number of cell within decomposition

Finally, we may also inquire which processor (rank) owns a certain point in the grid. Again, this requires a number of arguments that depends on the dimensionality of the grid.

```
int CHN_Point_owner(int decomposition, ...)
integer function CHN_POINT_OWNER(decomposition, index0, index1, ...)
   integer index0, index1, ...
```
Error return value: -1

IN	decomposition	handle to decomposition data structure
IN	index0	first grid point index
IN	index1	second grid point index
IN	...	subsequent grid point indices

3.4 Distributions

Once a decomposition is formed, a distributed variable (distribution) of a certain data type can be associated with it. In the current release the following elementary MPI data types are allowed: `MPI_REAL`, `MPI_DOUBLE_PRECISION`, `MPI_COMPLEX`, `MPI_INTEGER`, `MPI_CHARACTER` (corresponding to Fortran data types), `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_INT`, and `MPI_CHAR` (corresponding to C data types).

The user also specifies the tensor rank (number of tensor indices) of the variable, plus for each tensor index the number of components. For example, setting `rank` equal to two, and the first and second tensor index sizes equal to two and three, respectively, defines a (2×3) matrix at each point of the grid. A rank equal to zero signifies a scalar field variable.

To accommodate stencil operations the user specifies a nonnegative number of `ghost_points` that form a border of overlap points (communication buffer) around each cell (see Figure 3.5 on page 32).

The user also supplies the `start_address`, which refers to a range of memory locations reserved for storage of elements of type `data_type`. In Fortran 77, the starting address is often the beginning of a user declared array. In C it can be the same, but the null pointer can be used as a placeholder. Space can later be (re)assigned to the distributed variable through `CHN_Set_start_address`. In all cases it is user allocated space that is reserved.

```
int CHN_Create_distribution(int *distribution, int decomposition,
                      MPI_Datatype data_type, void *start_address,
                      int nghost, int rank, ...)
CHN_CREATE_DISTRIBUTION(distribution, decomposition, data_type, start_address,
                      nghost, rank, size0, size1, ..., ierr)
```

```
   integer distribution, decomposition, data_type, nghost, rank, size0, size1,
                        ..., ierr
   <type> start_address(*)
```

OUT	distribution	handle to distribution data structure
IN	decomposition	handle to decomposition data structure
IN	data_type	elementary MPI data type
IN	start_address	start in memory of local data storage
IN	nghost	thickness of border of ghost points
IN	rank	rank of tensor at each grid point
IN	size0	extent of first tensor index
IN	size1	extent of second tensor index
IN	...	extent of subsequent tensor indices

```
int CHN_Set_start_address(int distribution, void *start_address)
CHN_SET_START_ADDRESS(distribution, start_address)
   integer distribution
   <type> start_address(*)
```

INOUT	distribution	handle to distribution data structure
IN	start_address	start in memory of local data storage

The distribution variable is destroyed, and its handle reset to zero, by the function `CHN_Delete_distribution`.

```
int CHN_Delete_distribution(int *distribution)
CHN_DELETE_DISTRIBUTION(distribution, ierr)
   integer distribution, ierr
```

INOUT	distribution	handle to distribution data structure

The distribution data structure simply provides a structuring interpretation of space pointed to by the user. This makes it possible to operate on distributed data in any way the programmer deems convenient, through Charon functions, through plain Fortran or C, or—most often—both.

By default, all cells take up an equal amount of space. The layout is consistent with a storage declaration that allocates to all cells subarrays of identical dimensions. This makes it easy in both C and Fortran to declare a distributed variable of rank r on a grid of dimension n as an $(n + r + 1)$-dimensional array, where one dimension is of the size of the number of cells owned by each processor.

As an example we show the data layout in memory of a two-dimensional, scalar distributed variable (`rank = 0`) on the grid corresponding to that of Figure 3.1 on page 21, with a layer of ghost points of unit thickness[1]. Assume that the two cells whose grid points are indicated with solid disks (•) in Figure 3.5, are owned by the processor whose storage pattern we wish to examine. The lower left cell has *local* index zero, and the upper right cell has *local* index one. Ghost points for these cells are marked with open disks (○). The largest cell in the decomposition has dimensions (6×5), and this determines the default size of the subarrays associated with each cell. Since all cell subarrays, including that for the largest cell, need to accommodate the

[1] Henceforth, we speak of a 'distribution with `d` ghost points' if the thickness of its layer of ghost points is `d`.

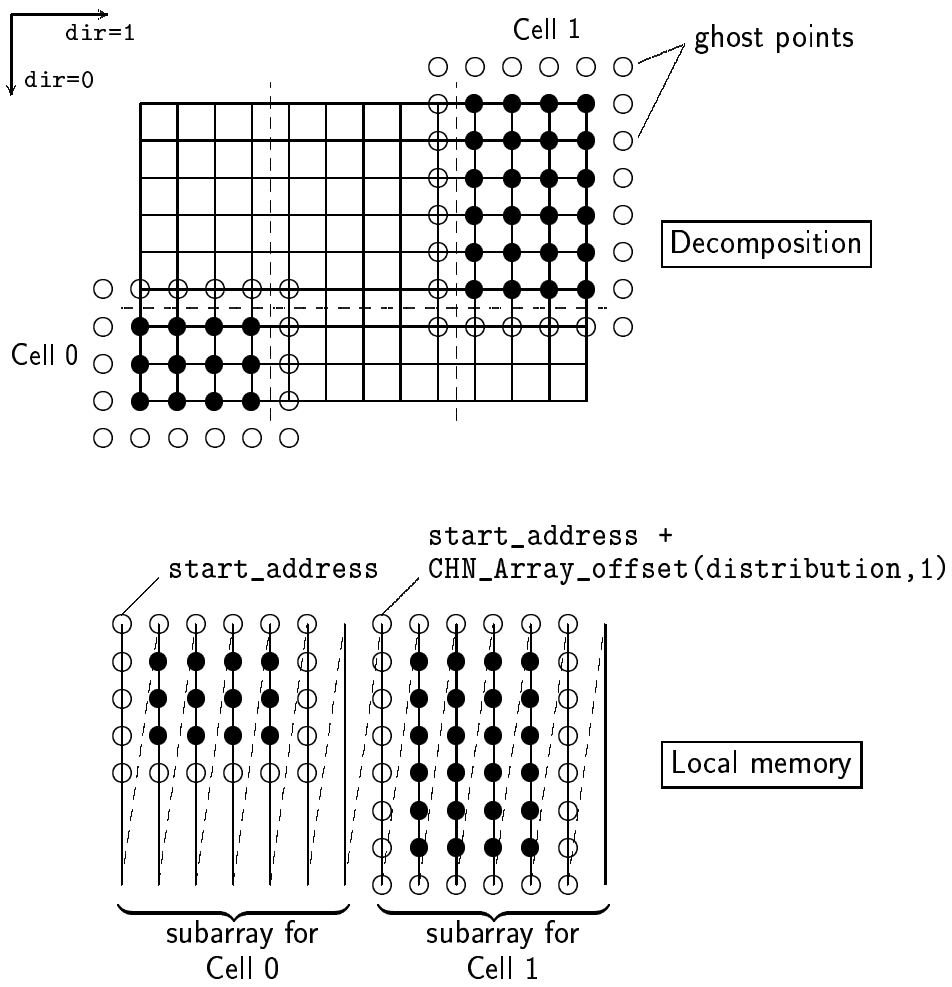


Figure 3.5: Data layout for sample scalar two-dimensional distributed variable

ghost points, the uniform subarray dimensions must be (8×7), even though less space suffices, in principle, for the smaller cells.

The continuous, snaking line in Figure 3.5 suggest how the linear address space in local memory can be viewed as a concatenation of two-dimensional subarrays, each corresponding to the memory required to store the data for one cell plus its ghost point values. This is the memory model used by Charon. Notice that the first coordinate in the grid corresponds to the fastest changing index in memory. The offset that determines where the second subarray starts relative to the beginning of the first, is computed by Charon, and can be obtained through a query function (see section 3.4.1).

The defaults for Charon memory use can be changed in several ways. The easiest is through the function `CHN_Compact_distribution`, which stipulates that only the minimum amount of space required for storing the data relating to the distributed variable should be used. Hence, dimensions of subarrays used for storing cell data will generally no longer be uniform across processors and cells. As a consequence, offsets for subarrays will also no longer be uniform, and must be computed by the user, or obtained through query functions.

```
int CHN_Compact_distribution(int distribution)
```

```
CHN_COMPACT_DISTRIBUTION(distribution, ierr)
   integer distribution, ierr
```

INOUT distribution	handle to distribution data structure

More control over memory usage is got by specifying explicitly where each cell c (where $0 \leq$ c $<$ CHN_Total_num_owned_cells) starts in memory, relative to the global starting address (CHN_Set_cell_array_offset), and what the *spatial* array dimensions are related to each cell (CHN_Set_cell_array_size). All offsets and sizes are in units of the basic data type of the distribution. It is legal to specify cell offsets that would let the data for different cells on the same processor overlap, or perhaps even completely coincide (useful for work arrays). However, the individual subarray dimensions cannot be made any smaller than is necessary to span the entire cell plus ghost points; only positive array padding is allowed. Whenever a subarray dimension is changed, the difference in the total size occupied by cell subarray is computed, and the subarrays following it in memory are shifted by that amount.

```
int CHN_Set_cell_array_offset(int distribution, int offset, int c)
CHN_SET_CELL_ARRAY_OFFSET(distribution, offset, c, ierr)
   integer distribution, offset, c, ierr
```

INOUT	distribution	handle to distribution data structure
IN	offset	relative starting address of call subarray
IN	c	local sequence number of cell within processor

```
int CHN_Set_cell_array_size(int distribution, int dir, int size,
                            int c)
CHN_SET_CELL_ARRAY_SIZE(distribution, dir, size, c, ierr)
   integer distribution, dir, size, c, ierr
```

INOUT	distribution	handle to distribution data structure
IN	dir	coordinate direction
IN	size	array size of cell subarray (includes ghost points)
IN	c	local sequence number of cell within processor

When the tensor rank of the distribution is nonzero, the user may specify if the tensor indices are the fastest varying components in memory (CHN_Set_tensor_indices_first; the default), or if it is the spatial coordinates (CHN_Set_tensor_indices_last). For example, a user may specify in Fortran that an array of shear stress coefficients should be dimensioned var(3,3,nx,ny,nz) on a three-dimensional grid, in which case the tensor rank indices are grouped in clusters of nine adjacent memory locations; they come first. Moreover, the tensor indices start with index one, which is specified using CHN_Set_tensor_start_index. Alternatively, the programmer may want to write the vector of physical quantities density, x-momentum, y-momentum, and energy on a two-dimensional grid as var(nx,ny,4), in which case all densities for a given cell are in adjacent memory locations, followed by all x-momenta, etc.; the tensor rank indices come last.

Often the large majority of arrays in an application are of one type of tensor rank index ordering and numbering. In this case it is convenient to use the "all" variations of the functions specifying the tensor index position. Their settings can be overridden for individual distributions.

It is important to recognize that the "all" variations influence the content of only those Charon variables that are defined subsequently.

As will be discussed in Section 6.3, subsets of tensors components may be identified and selected for manipulation. These subsets, or masks, are created, modified, and queried by specialized functions. However, there is an overlap between distribution and mask manipulation functions in the area of tensor component specifications. Hence, two of the functions defined below (CHN_Set_tensor_start_index and CHN_Set_all_tensor_start_indices) can also be applied to tensor masks. In fact, the latter is implicitly active for all subsequent definitions of distributions and tensor masks alike.

```
int CHN_Set_tensor_start_index(int dist_or_mask, int index)
CHN_SET_TENSOR_START_INDEX(dist_or_mask, index, ierr)
   integer dist_or_mask, index, ierr
```

INOUT	dist_or_mask	handle to distribution or tensor mask data structure
IN	index	starting value for tensor rank indices

```
int CHN_Set_tensor_indices_first(int distribution)
CHN_SET_TENSOR_INDICES_FIRST(distribution, ierr)
   integer distribution, ierr

int CHN_Set_tensor_indices_last(int distribution)
CHN_SET_TENSOR_INDICES_LAST(distribution, ierr)
   integer distribution, ierr
```

INOUT	distribution	handle to distribution data structure

```
int CHN_Set_all_tensor_start_indices(int index)
CHN_SET_ALL_TENSOR_START_INDICES(index, ierr)
   integer index, ierr
```

IN	index	starting value for tensor rank indices

```
int CHN_Set_all_tensor_indices_first(void)
CHN_SET_ALL_TENSOR_INDICES_FIRST(ierr)
   integer  ierr

int CHN_Set_all_tensor_indices_last(void)
CHN_SET_ALL_TENSOR_INDICES_LAST(ierr)
   integer  ierr
```

Certain specialized operations on the indices, such as overindexing for vector computing, and index reduction of higher-dimensional variables, are described in Chapter 9.

All distribution creation, manipulation, and destruction operations are local and collective, which means that all processors in the corresponding communicator must call these routines with the same parameters (except those that associate storage space on individual processors).

3.4.1 Distribution query functions

The functions CHN_Decomposition, CHN_Datatype, CHN_Storage_space, CHN_Num_ghost_points, CHN_Tensor_rank, CHN_Tensor_start_index, and CHN_Tensor_size return for a given distribution: the decomposition on which it is based, the elementary data type, the total amount of memory (in units of the elementary data type) required to store the data for the distributed variable, the number of ghost points surrounding each cell, the tensor rank, the starting index of the tensor components, and—for a a particular index of the rank—the number of components, respectively. The last three functions can also be invoked to query so-called tensor masks (Section 6.3.1). The function CHN_Start_address returns, in C, a pointer to the actual address where the storage space associated with the data of the distributed variable starts. In Fortran 77, where pointers are not possible return values of functions, the result is cast to ADDRESSTYPE—often INTEGER—that is the same size as a C void pointer (See Section 1.6.2). The function CHN_Datatype does not have an obviously invalid return value to be used in case of an incorrect input parameter. We opt to return the MPI data type MPI_BYTE, which cannot be used as a data type of a distributed variable in Charon.

```
int CHN_Decomposition(int distribution)
integer function CHN_DECOMPOSITION(distribution)
   integer distribution
```
Error return value: -1

```
int CHN_Storage_space(int distribution)
integer function CHN_STORAGE_SPACE(distribution)
   integer distribution
```
Error return value: -1

```
int CHN_Num_ghost_points(int distribution)
integer function CHN_NUM_GHOST_POINTS(distribution)
   integer distribution
```
Error return value: -1

```
void* CHN_Start_address(int distribution)
ADDRESSTYPE function CHN_START_ADDRESS(distribution)
   integer distribution
```
Error return value: 0

```
MPI_Datatype CHN_Datatype(int distribution)
integer function CHN_DATATYPE(distribution)
   integer distribution
```
Error return value: MPI_BYTE (not useable in Charon)

```
int CHN_Tensor_rank(int dist_or_mask)
integer function CHN_TENSOR_RANK(dist_or_mask)
   integer dist_or_mask
```
Error return value: -1

```
int CHN_Tensor_start_index(int dist_or_mask)
integer function CHN_TENSOR_START_INDEX(dist_or_mask)
   integer dist_or_mask
```
Error return value: greatest representable positive integer

IN	distribution	handle to distribution data structure
IN	dist_or_mask	handle to distribution or tensor mask data structure

```
int CHN_Tensor_size(int dist_or_mask, int tensor_index)
integer function CHN_TENSOR_SIZE(dist_or_mask, tensor_index)
   integer dist_or_mask, tensor_index
```
Error return value: -1

IN	dist_or_mask	handle to distribution or tensor mask data structure
IN	tensor_index	sequence number of tensor index

To gain access to and obtain proper dimensioning parameters for the subarrays used to store the data associated with the distributed variable, we can use the functions `CHN_Cell_array_offset` and `CHN_Cell_array_size`. These determine the offset of the cell data from the start address of the distributed variable, and the dimensions of the cell data subarray in a particular coordinate direction, respectively.

```
int CHN_Cell_array_offset(int distribution, int c)
integer function CHN_CELL_ARRAY_OFFSET(distribution, c)
   integer distribution, c
```
Error return value: Greatest representable negative integer

IN	distribution	handle to distribution data structure
IN	c	local sequence number of cell within processor

```
int CHN_Cell_array_size(int distribution, int dir, int c)
integer function CHN_CELL_ARRAY_SIZE(distribution, dir, c)
   integer distribution, dir, c
```
Error return value: -1

IN	distribution	handle to distribution data structure
IN	dir	coordinate direction
IN	c	local sequence number of cell within processor

To find out what the default starting value is for all tensor rank indices for all distributions, use the following.

```
int CHN_All_tensor_start_indices(void)
integer function CHN_all_tensor_start_indices()
```
Error return value: none

Probe functions that check whether tensor rank indices are the fastest (first) varying components in memory of the distributed variable return 1 when true, 0 when false. Similarly defined functions test whether tensor rank indices are the slowest (last) varying components in memory of the distributed variable.

```
int CHN_Tensor_indices_first(int distribution)
integer function CHN_TENSOR_INDICES_FIRST(distribution)
   integer distribution
```
Error return value: -1

```
int CHN_Tensor_indices_last(int distribution)
integer function CHN_TENSOR_INDICES_LAST(distribution)
   integer distribution
```
Error return value: -1

IN	distribution	handle to distribution data structure

```
int CHN_All_tensor_indices_first(void)
integer function CHN_ALL_TENSOR_RANKS_FIRST()
```
Error return value: none

```
int CHN_All_tensor_indices_last(void)
integer function CHN_ALL_TENSOR_INDICES_LAST()
```
Error return value: none

3.4.2 Local storage details

Storage for the distributed variable shown in Figure 3.5 on page 32 can be suitably dimensioned in Fortran through

```
dimension var(8,7,2)
```

or in C through

```
<type> var[2][7][8];
```

where <type> signifies the appropriate data type. Thus, if the grid point starting indices for cell c in coordinate direction dir are given by start(dir,c) (this index array can be filled using the query function CHN_Cell_start_index), then the data item of Fortran array var(i,j,c) corresponds to grid point (i+start(0,c)-1,j+start(1,c)-1).

In general, if an n-dimensional scalar distribution has nghost ghost points and if its storage space has been dimensioned in Fortran through

```
dimension var(max_0, max_1, ..., max_{n-1})
```

or in C through

```
<type> var[max_{n-1}]...[max_1][max_0];
```

with $\texttt{max}_i$ being 2*nghost plus the maximum cell size in coordinate direction i, then "var($\texttt{i}_0$+1, $\texttt{i}_1$+1,...,$\texttt{i}_{n-1}$+1,c+1)" (Fortran) and "var[c][$\texttt{i}_{n-1}$]...[$\texttt{i}_1$][$\texttt{i}_0$]" (C) both correspond to the value of the distributed variable at grid point ($\texttt{i}_0$+start(0,c)-nghost,$\texttt{i}_1$+start(1,c)-nghost,...,$\texttt{i}_{n-1}$+start(n-1,c)-nghost). Here the array start(dir,c) has the same meaning as above.

An important consequence of this indexing scheme is that if the decomposition contains just a single cell without ghost points, and if the storage space has been dimensioned in Fortran through

```
dimension var(start(0,0)  :start(0,0)  +max0,
              start(1,0)  :start(1,0)  +max1,
              ...,
              start(n-1,0):start(n-1,0)+maxn-1)
```

then `var(i0,i1,...,in-1,c)` corresponds to the value of the distributed array at grid point `(i0,i1,...,in-1,c)`. In other words, the same indices can be used for accessing values of distributed variables through Charon, or through the local array. In C the default array starting index cannot be changed from zero, but usually macros are employed to mimic indexing in multi-dimensional arrays of variable size, in which case indexing is as flexible as in Fortran.

In legacy codes this fact can be exploited by 'reverse engineering' of the Charon grid. If a scalar variable relating to a discretization grid has been dimensioned `var(nx,ny,nz)` in Fortran, then we can define a Charon grid of the same dimensions and set its starting indices to one. Subsequently, we create a solo-partition section and, based on that, a solo-partition decomposition. Finally, we associate with the decomposition a scalar distribution without ghost points. This establishes the identity relation between Charon and local array indexing. Consequently, as will be demonstrated in Chapter 4, manipulations and assignments involving array `var` can switch transparently between Charon and the local array. The convenience of this mechanism is that it allows partial conversion of a code using Charon constructs, while leaving the rest of the code entirely unchanged.

3.5 Distribution examples

We give some examples, in C and in Fortran, of how Charon can be used to design distributed implementations of data structures for practical serial scientific computing programs on structured grids. As a notational convenience, the names of the distributions in the examples are derived from those of the data arrays they represent by appending an underscore ('_') character to the array name.

3.5.1 Staggered grids

Staggered grids are a common and useful instrument in fluid flow computations. They allow the numerical analyst to specify certain quantities at some locus of the grid (for example, all physical quantities at the center of cells), and others at other loci (for example, grid metrics at cell vertices). Figure 3.6 gives an example of a two-dimensional (2D) staggered grid. Physical quantities, indicated by solid disks (•), are stored at cell centers, and metric quantities, indicated by diamonds (⋄), are stored at the cell vertices.

The staggered computational grid embodies two sets of array indices. One to select physical components, and one to select metrics components. This is reflected in the definition of two Charon grids that differ in their grid sizes by one unit in each coordinate direction. Because there is no one-to-one correspondence between the points in both Charon grids, we also cannot place cuts such that there is a pointwise correspondence between the cells of their Cartesian sections. However, we do insist on *consistent* cuts, meaning that if one cell of the metrics section contains the metrics coefficients for the left strip of points in the physics section's cell, then all others should as well. This is accomplished by copying the cuts of the metrics section to that of the

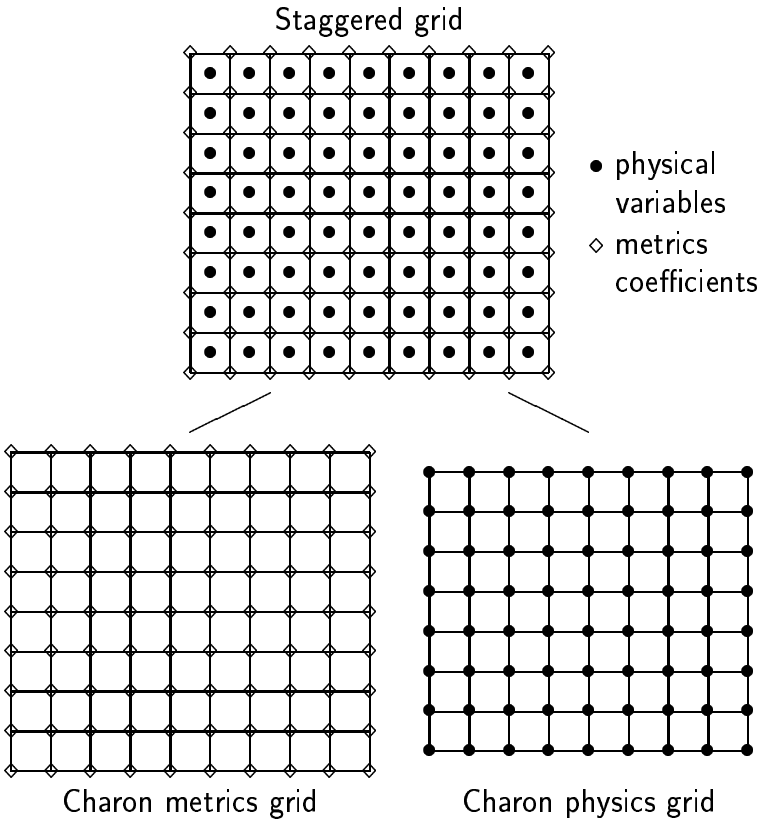


Figure 3.6: Staggered computational grid uses two Charon grids

physics section, with a constant shift. The fragment of C code below implements this method for a multi-partition decomposition, but the technique applies to any valid decomposition.

```
int m_grid, m_section, m_decomposition, ph_grid, ph_section,
    ph_decomposition, cut, dir, npoints[2];

  for (dir=0; dir<2; dir++) scanf("%d",&npoints[dir]);
  CHN_Create_grid(&m_grid,MPI_COMM_WORLD,2);
  CHN_Create_grid(&ph_grid,MPI_COMM_WORLD,2);
  for (dir=0; dir<2; dir++)
    CHN_Set_grid_size(m_grid,dir,npoints[dir]);
    CHN_Set_grid_size(ph_grid,dir,npoints[dir]-1);

  CHN_Create_section(&m_section,m_grid);
  CHN_Create_section(&ph_section,ph_grid);
  CHN_Set_multipartition_cuts(m_section);
  for (dir=0; dir<2; dir++)
    CHN_Set_num_cuts(ph_section,dir,CHN_Num_cuts(m_section,dir));
    /* copy cuts between sections, with constant shift 0 */
    for (cut=0; cut<CHN_Num_cuts(ph_section,dir); cut++)
      CHN_Set_cut(ph_section,dir,cut,CHN_Cut(m_section,dir,cut));
```

```
CHN_Create_decomposition(&m_decomposition,m_section);
CHN_Create_decomposition(&ph_decomposition,ph_section);
CHN_Set_multipartition_owners(m_decomposition);
CHN_Set_multipartition_owners(ph_decomposition);
```

Notice that although the cuts were placed 'by hand' for the physics section, we can still use the `CHN_Set_multipartition_owners` for both derived decompositions, and be assured that corresponding cells have the same owners. That is because the logical layout of cells is identical, even though the exact cell sizes in the decompositions may differ.

3.5.2 Multi-dimensional Fast Fourier Transform

The multi-dimensional Fast Fourier Transform (FFT) is a popular technique for obtaining high-accuracy difference schemes on structured grids. It is a global method in which a (vector of) coefficients at each grid point is computed from data at all other grid points. All coefficients can be computed independently, in principle, which makes FFT's naturally data parallel, but expensive. On a grid with N points the amount of work is proportional to N^2 if all coefficients are computed separately. Moreover, on a distributed computer a vast amount of data needs to flow between processors.

Multi-dimensional FFT's can be sped up by breaking them into sequences of one-dimensional (1D) FFT's, one for each grid line. Within the grid line points can share many common subexpressions, rendering the final method as efficient as $\mathcal{O}(N \log N)$. Unfortunately, sharing subexpressions means that there is still a large amount of data traffic between all points on a grid line, which argues against dividing grid lines among processors. Since multi-dimensional FFT's require 1D FFT's in all coordinate directions of the grid, it would appear that no domain decomposition can be used.

More precisely, not a *single* domain decomposition can be constructed that is efficient for 1D FFT's in all coordinate directions. But the grid may be divided in different ways, depending on the orientation of the 1D grid lines, and data may be 'transposed' (`CHN_Redistribute`; see Section 6.1.2) between these different distributions. Figure 3.7a shows a schematic of an example of two different slicewise decompositions of the grid that together are sufficient to accommodate 3D FFT's, provided the grid is large enough. If the number of processors exceeds the number of grid points on all sides of the grid, a higher-dimensional decomposition must be used. The three pencilwise decompositions shown in Figure 3.7b offer more parallelism.

Consider now the sample fragment of Fortran code below, used to set up the data structures for a 3D FFT problem. The dimensions of the discretization grid are read from standard input, and are stored in the array `npoints`. We will assume that they are all powers of two, so that the 1D FFT's can be computed efficiently. `Nmax` is sufficiently large to accommodate any of the expected inputs. There is one Fourier coefficient per grid point, represented by the scalar array `coef`. Althrough we create three different distributions to represent this array, we reserve just two times its size, because we expect never to need more than two active distributions at the same time. In the next release of Charon *in situ* redistributions will be accommodated, in which case no additional space needs to be claimed. We assume that no ghost points are needed (if there were, more space should be allocated).

```
integer grid, section(3), decomposition(3), coef_(3), npoints(3), dir, ierr
```

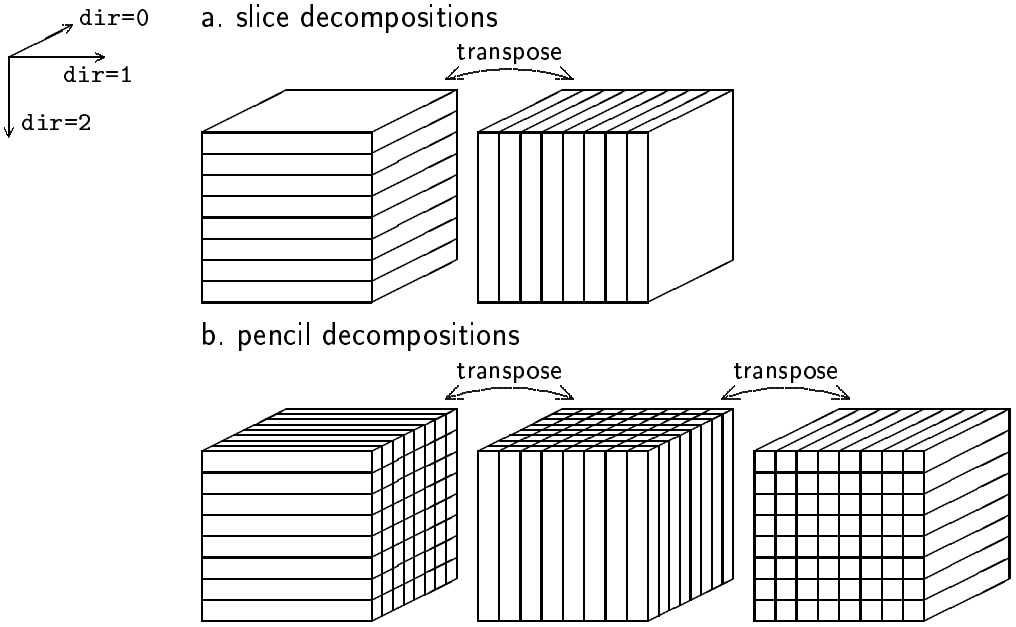


Figure 3.7: Dynamic decompositions for three-dimensional FFT's

```
  real coef(nmax,nmax,nmax,2)

c create and initialize the grid
  call CHN_CREATE_GRID(grid,MPI_COMM_WORLD,3,ierr)
  read *, (npoints(dir),dir=1,3)
  do  dir = 1, 3
    call CHN_SET_GRID_START_INDEX(grid,dir-1,1,ierr)
    call CHN_SET_GRID_SIZE(grid,dir-1,npoints(dir),ierr)
  end do

c create three different sections, decompositions and distributions
c for different pencil distributions
  do  dir = 1, 3
    call CHN_CREATE_SECTION(section(dir),grid,ierr)
    call CHN_EXCLUDE_PARTITION_DIRECTION(section(dir),dir-1,ierr)
    call CHN_SET_UNIPARTITION_CUTS(section(dir),CHN_EVEN_CUTS,ierr)

    call CHN_CREATE_DECOMPOSITION(decomposition(dir),section(dir),ierr)
    call CHN_SET_UNIPARTITION_OWNERS(decomposition(dir),ierr)

    call CHN_CREATE_DISTRIBUTION(coef_(dir),decomposition(dir),MPI_REAL,
 $                               coef(1,1,1,1),0,0,ierr)
  end do

c reset the starting addresses of the second and third distributions.
c the third equals that of the first
    call CHN_SET_START_ADDRESS(coef_(2),coef(1,1,1,2),ierr)
```

```
call CHN_SET_START_ADDRESS(coef_(3),coef(1,1,1,1),ierr)
```

Observe that the number of processors P does not enter (this part of) the program. The shapes of the pencils will be computed automatically to minimize their surface areas. If the program is run on one processor, the total data will still fit in in the space allocated to array `coef`. If the number of processors is a square, each divided coordinate direction receives the same number of cuts, and Charon's deterministic cutting algorithm also guarantees that the cuts in the same coordinate direction on different sections coincide exactly in space. This is a performance benefit for the `CHN_Redistribute` function, because relatively few communications are required to establish a data transposition among the processors. If the number of processors is not a square, there is a possibility of a mismatch of the distributions.

For example, assume that $P = 12 (= 3 \times 4)$. In that case `coef_(1)` will receive 2 cuts in the second and 3 in the third coordinate direction. `Coef_(2)` will receive 2 cuts in the first and 3 in the third coordinate direction, and `coef_3` will receive 2 cuts in the first and three in the second coordinate direction. Any redistribution between `coef_(1)` and `coef_(2)` and between `coef_(2)` and `coef_(3)` takes place efficiently, but if the user would want to redistribute between `coef_(1)` and `coef_(3)`, the thicknesses of the pencils would not be compatible. This would not lead to program errors, but would make the redistribution less efficient. In our sample program this situation is not likely to occur, because we have already ruled out redistributions between `coef_(1)` and `coef_(3)` by allowing their storage spaces to overlap.

3.5.3 Three-dimensional grid with two-dimensional scratch array

Many scientific computing programs that solve problems in three (or more) space dimensions can benefit from the use of lower-dimensional scratch arrays. They save memory, and can also improve cache performance due to improved data locality. Charon readily supports the use of grids of different dimensionality within the same application. But a complication arises if the programmer insists that a certain element of a scratch array be owned by the same processor that owns the corresponding point in the higher-dimensioned grid. Concretely, the problem is how to associate an array element with just two indices (e.g. `var(i,j)`) to a grid point that has three indices (and a unique owner processor).

Charon provides two solutions. The first is to define a distribution based on the 3D grid, but to 'freeze' one of its indices at the value that corresponds to the particular slice of the grid for which the scratch array is used. This technique is demonstrated in Section 9.2. This method results in the same notational convenience of fewer array indices, but actually uses the entire 3D space to store data, and hence does not save memory nor improves cache performance. Moreover, only a contiguous subset of indices, starting from the highest, can be frozen thusly.

The second solution requires the user to define one 2D grid and section, and possibly several 2D decompositions and distributions—but an amount of space equal to at most just one slice of the grid. If every point of `var(i,j)` is owned by the same processor, regardless of the slice of the grid, the problem essentially utilizes a pencil decomposition, and a single 2D decomposition and distribution for the scratch array will suffice. This is indicated in Figure 3.8a, where the dropped index from the 2D work array corresponds to the undivided axis of the grid.

If the ownership of point `var(i,j)` does depend on the particular slice of the grid, such as indicated in Figure 3.8b, then it is easiest (although a little redundant) to define a separate

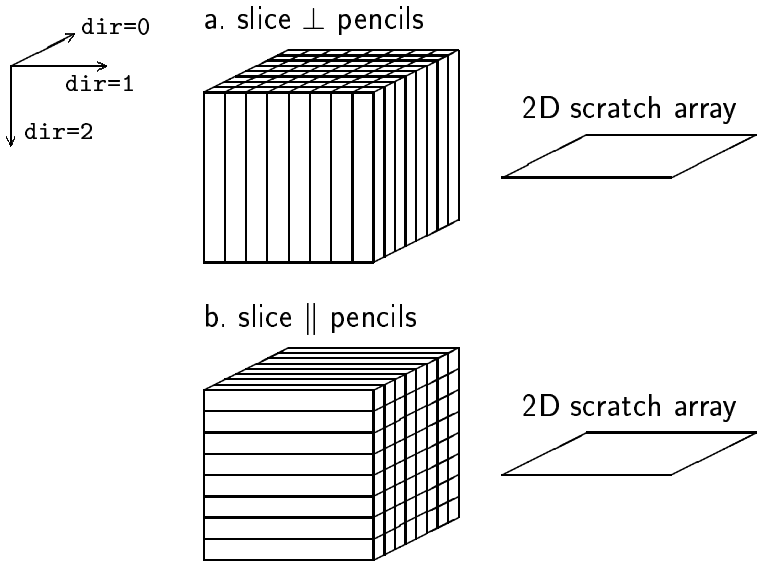


Figure 3.8: Lower-dimensional scratch arrays

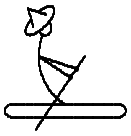
decomposition and distribution for each slice. This is exemplified by the Fortran code below. We use the multi-partition decomposition for this computation, because a uni-partition would result in a poor load balance when working on the scratch array.

```
  integer grid3d, section3d, decomposition3d, grid2d, section2d,
 $        decomposition2d(KMAX), work2d_(KMAX), npoints(3), dir, ierr
  double precision work2d(WORK2DMAX)

c create and initialize the grids
  read *, (npoints(dir),dir=1,3)
  call CHN_CREATE_GRID(grid2d,MPI_COMM_WORLD,2,ierr)
  do  dir = 1, 2
    call CHN_SET_GRID_START_INDEX(grid2d,dir-1,1,ierr)
    call CHN_SET_GRID_SIZE(grid2d,dir-1,npoints(dir),ierr)
  end do

  call CHN_CREATE_GRID(grid3d,MPI_COMM_WORLD,3,ierr)
  do  dir = 1, 3
    call CHN_SET_GRID_START_INDEX(grid3d,dir-1,1,ierr)
    call CHN_SET_GRID_SIZE(grid3d,dir-1,npoints(dir),ierr)
  end do

c create the sections, copying cuts from the 3D section to 2D
  call CHN_CREATE_SECTION(section3d,grid3d,ierr)
  call CHN_EXCLUDE_PARTITION_DIRECTION(section3d,0,ierr)
  call CHN_SET_MULTIPARTITION_CUTS(section3d,ierr)

  call CHN_CREATE_SECTION(section2d,grid2d,ierr)
  call CHN_SET_NUM_CUTS(section2d,CHN_num_cuts(section3d,1),1,ierr)
```

```
  call CHN_SET_EVEN_CUTS(section2d,1,ierr)

c create the decompositions, copying ownership from 3D to 2D
  call CHN_CREATE_DECOMPOSITION(decomposition3d,section3d,ierr)
  call CHN_SET_MULTIPARTITION_OWNERS(section3d,ierr)

  do   k = 1, npoints(3)
    call CHN_CREATE_DECOMPOSITION(decomposition2d(k),section2d,ierr)
    do strip = 1, CHN_NUM_CELLS(decomposition2d(k),1)
      cell2d = CHN_CELL_INDEX(decomposition2d(k),0,strip-1)
      owner3d = CHN_POINT_OWNER(decomposition3d,
 $                   CHN_CELL_START_INDEX(decomposition2d(k),0,cell2d),
 $                   CHN_CELL_START_INDEX(decomposition2d(k),1,cell2d),k)
      call CHN_SET_CELL_OWNER(decomposition2d(k),owner3d,cell2d,ierr)
    end do
  end do

c create the 2D distributions; they all have the same starting address
  do   k = 1, npoints(3)
    call CHN_CREATE_DISTRIBUTION(work2d_(k),decomposition2d(k),
 $                               MPI_DOUBLE_PRECISION,work2d,0,0,ierr)
  end do
```

We note that the way the 2D work arrays are defined in this example gives the correct result for any 3D decomposition, not just the 3D pencil decomposition shown in Figure 3.8b.

Chapter 4



Distributed execution support tools

It is useful to define a distinction between distributed and parallel scientific computing. By distributed computing we mean performing operations on a data set that resides on multiple computers. Usually, each of these computers will contain only part of that data set, although often there will be some overlap between the subsets. Parallel computing is the simultaneous execution of several processes related to the same computational task. Charon's aim is to obtain a distributed, parallel scientific application.

The previous chapter has shown how data can be distributed. In this chapter we explain how operations can be performed on the distributed data set. The result is a distributed program, but not yet a parallel program. The next chapter will explain how a Charon distributed program is gradually turned into a high-performance parallel program. The reason for this multi-step approach is to obtain programmability. Distributing the application requires many textual changes to the program, as we will see, but virtually all are very straightforward. No change in program structure or logic is required when converting a nondistributed program or design to a distributed environment. The more difficult task of parallelizing the distributed program can subsequently be undertaken one routine, loop, or statement at a time. This makes developing and debugging the parallel program easier than before, because at all times the programmer maintains a correct program.

The distribution of the computational tasks hides from the user all such bookkeeping details as domain decomposition, local and remote data, communication, etc. At this highest level of Charon's abstraction, computational efficiency is generally very poor, but that is not a problem. In the parallelization phase, at the next lower level of abstraction, performance improvements are obtained, again making use of Charon tools.

Charon does not provide automatic code conversion. All distribution and parallelization is carried out by the user, who retains complete control over data lay-out and program flow. Hence, the necessary code changes must be kept at a minimum. For that purpose Charon offers distributed execution support tools that simulate a single data space and a single thread of control. Assignments and control structures are exact images of the serial program, and the resulting code is executed by all processors; Charon simulates a single serial program, or, more accurately,

a single replicated program counter.

We use the following rationale for the implementation. Each element of a distributed variable is associated with a grid point, which has a unique owner processor, as was detailed in the previous chapter. So it is most natural—and often least expensive—to employ an *owner-assigns* rule (a variation of the oft-cited owner-computes rule): whenever an element of a distributed variable occurs on a left-hand side of an assignment, the processor who owns it is responsible for its update. Since all processors execute the same code within the same control structure, we have to provide a mechanism to skip assignments to nonlocal memory locations; the replicated program counter 'pauses' on all processors, except on the one executing the local assignment, and 'resumes' collectively immediately thereafter.

4.1 Elementwise distributed execution

The implementation is as follows. Each assignment in the serial program is converted into a call to `CHN_Assign`, which takes as arguments a left hand side (an address) and a right hand side (a value). If the address does not exist on the calling processor, signified by the null value (an illegal, nonreachable address), the assignment is effectively ignored; it is masked. Masking is provided by the function `CHN_Address`, which returns an actual memory location for a local element of a distributed variable, and null otherwise. Assignment masking alone, however, is not sufficient, since assembly of the right hand side of the assignment may involve contributions from local as well as (possibly several) remote memories. For this purpose the generic function `CHN_Value`—specializations will be described below—is introduced. It operates on distributed variables and always returns the proper value, regardless of which processor owns the donor point. Analogous to the *owner-assigns* rule for left hand sides of assignments, we apply the *owner-serves* rule to the right hand side components, meaning that the unique processor that owns a particular grid point is responsible for producing values associated with the point upon request by `CHN_Value`.

No distinction is made between values returned by the function `CHN_Value` and values of nondistributed variables and constants. All are *rvalues* (*r*ight hand side *values*) in C terminology. Similarly, no distinction is made between addresses returned by the function `CHN_Address`, and addresses of nondistributed variables[1]. Both are *lvalues* (*l*eft hand side *values*). In a correct program using only top-level Charon tools, rvalues always exist, whereas lvalues are defined only if they are local. Alternatively, we may say that the highest level of abstraction of Charon only offers (implicitly invoked) remote gets, not puts.

Syntax of global access functions:

```
int CHN_Assign(void *my_address, <type> my_value)
CHN_ASSIGN(my_address, my_value, ierr)
   integer     ierr
   ADDRESSTYPE my_address
   <type>      my_value
```

INOUT	my_address	address of local element of distribution
IN	my_value	value to be assigned to local element of distribution

[1] But see Section 4.4 for a discussion of the application of `CHN_Assign` to an address that is not obtained through `CHN_Address`.

```
<type> CHN_<type>_value(int distribution, int sbs0, int sbs1, ...)
<type> function CHN_<type>_VALUE(distribution, sbs0, sbs1, ...)
   integer distribution, sbs0, sbs1, ...
```
Error return value: largest representable value, or EOF (for characters)

IN	distribution	handle to distribution data structure
IN	sbs0	first subscript of distribution (coordinate or tensor index)
IN	sbs1	second subscript of distribution (coordinate or tensor index)
IN	...	subsequent subscripts

```
void   *CHN_Address(int distribution, int sbs0, int sbs1, ...)
ADDRESSTYPE function CHN_ADDRESS(distribution, sbs0, sbs1, ...)
```
Error return value: 0

IN	distribution	handle to distribution data structure
IN	sbs0	first subscript of distribution (coordinate or tensor index)
IN	sbs1	second subscript of distribution (coordinate or tensor index)
IN	...	subsequent subscripts

<type> can be any of Int, Float, Double, Char (C) INTEGER, REAL, DOUBLE_PRECISION, REAL8, COMPLEX, or CHARACTER (Fortran), which correspond to the MPI data types of the distributions. Observe that in Fortran the function CHN_Address returns an address stuffed in a variable of type ADDRESSTYPE—often INTEGER—that is the same size as a C void pointer (See Section 1.6.2). Note also that the CHN_Assign operator is overloaded; it can take values and addresses of any elementary types, as long as they are consistent, that is, my_value and my_address must refer to the same type.

A subtle side effect of the overloading is that there is no way that the compiler can check whether the value being assigned is properly typed. This is best illustrated as follows. If the assignment "r1(i,j) = 5" is encountered and if r1 has been declared a double precision array, then the integer 5 will automatically be promoted to a double precision number as well. However, if the Charon wrapper "call CHN_Assign(CHN_Address(r1_,i,j),5)" is used (with r1_ the integer handle of the distribution associated with array r1), there is not enough information to decide what, if any, promotion should be applied to the constant 5. So the value of this simple expression will be passed as an integer (often four bytes long) to CHN_Assign. If the latter is expecting an eight byte double precision number, an error will occur. The solution is to force the correct type of the rvalue, for example by using 5.0 or 5.0d0 instead of 5. We mark CHN_Assign's my_address parameter as INOUT, because the contents of the address may be changed, even though the address itself is not modified.

In C the implementation of the different types of my_value can be handled cleanly through the stdarg facility, which does not exist in Fortran. Hence, the above definition of CHN_ASSIGN is not formally possible in Fortran; it is simply a reflection of what the Fortran definition might look like, if a prototype would be written for it. In reality, virtually all Charon Fortran functions and procedures are immediate fall-throughs to C functions.

Subscripts sbs1, sbs2, ... are indices with respect to the global grid, as defined in the Charon grid variable, or the tensor indices of the distribution. For example, if a double precision

tensor T_ of rank two is defined on a four-dimensional grid, with the tensor indices positioned first (the default), then CHN_Address(T_,p,q,i,j,k,l) refers to the (p,q) tensor component at grid point (i,j,k,l). Similarly, if the tensor indices are positioned last, CHN_Double_value(T_,i,j,k,l,p,q) refers to the same component. In general, the subscripts are placed in order of increasing memory stride.

4.2 Blockwise distributed execution

Sometimes it is convenient to pass an element of an array variable to a procedure that operates on multiple elements adjacent to it in memory. For example, a user may want to call a fast assembly routine to compute the solution to a small, fixed-size matrix problem at each point.

The difficulty with this coding style is that the statements that do the actual work are in procedures that have no knowledge of the overall partitioned grid. They operate on addresses and neighboring memory locations that are passed through parameter lists. Moreover, it may be impossible to modify the assembly routine. But even if we would somehow be able to change it, the strategy of translating every assignment into a call to CHN_Assign would not work, because no global grid coordinates are available.

The solution is to demand that such procedures execute atomically, which means that there must be a single, known address (my_address) that acts as the start of a region of lvalues *on the same processor*. No other values may be modified within the same procedure. There may be several contiguous regions of rvalues, whose sizes must be known at run time. They are made available ('served') to the user-supplied procedure 'subf' through the function CHN_Mvalue, usually under the control of the Charon function CHN_Invoke.

```
int CHN_Invoke(void (*subf)(), void *my_address, int nr,
               void *my_values0, void *my_values1, ...)
CHN_invoke(subf, my_address, nr, my_values0, my_values1, ..., ierr)
   external subf
   integer nr, my_address, my_values0, my_values1, ..., ierr
```

IN	subf	user-supplied procedure
INOUT	my_address	start address of region of lvalues
IN	nr	number of regions of rvalues
IN	my_values0	start address of first region of rvalues
IN	my_values1	start address of second region of rvalues
IN	...	start address of subsequent regions of rvalues

```
void *CHN_Mvalue(n, distribution, sbs0, sbs1, ....)
ADDRESSTYPE function CHN_MVALUE(n, distribution, sbs0, sbs1, ....)
   integer n, distribution, sbs0, sbs1, ....
```

Error return value: 0

IN	n	number of contiguous data elements requested
IN	distribution	handle to distribution data structure
IN	sbs0	first subscript of distribution (coordinate or tensor index)
IN	idx1	second subscript of distribution (coordinate or tensor index)
IN	...	subsequent subscripts

CHN_Invoke examines the argument my_address, which determines which processor is responsible for the execution of the user-supplied procedure subf, based on the owner-assigns rule. All other processors skip the execution of subf, but cooperate in providing rvalues, as needed, through communications implicitly invoked by CHN_Mvalue. Again, atomicity is assumed, i.e. the processor that owns the first element of the distributed variable in the mvalue argument list is also responsible for supplying subsequent elements. If the first element is local, the action of CHN_Mvalue is nearly identical to that of CHN_Value. If remote, a communication requesting n data elements is initiated. In either case, upon completion the function CHN_Mvalue points to the start of a buffer region containing the requested values (address stuffed in a variable of type ADDRESSTYPE—usually an integer—in Fortran; see Section 1.6.2). The actual number of bytes transferred depends on the specific data type of the distributed variable. Naturally, space must be reserved on those processors receiving remote data in buffers, and these buffers can only be released when the programmer decides their contents are no longer needed. Section 9.5 explains how this is accomplished.

The subroutine subf is defined by the user as follows.

```
void subf(<type> *my_address, <type> *my_values0, ..., <type> *my_values_{nr-1});
SUBF(my_address, my_values0, ..., my_values_{nr-1})
  <type> my_address(*), my_values0(*), ..., my_values_{nr-1}(*)
```

Neither Fortran nor C provides information about the number of actual parameters with which a subroutine is called, so the user must indicate to CHN_Invoke the number of separate regions of input values through the parameter nr.

CHN_Assign, CHN_Address, and CHN_Invoke are always local. CHN_Value and CHN_Mvalue are collective and nonlocal by default. The next chapter will show how to change their default operation mode. Such mode changes correspond to the relaxation of the owner-assigns and owner-serves rules. The functions presented in this section are sufficient to move many serial programs based on structured grids to a distributed environment while retaining serial logic. They make up the bulk of Charon's top-level distributed execution support.

4.3 Serial consistency

Correctness (i.e. serial consistency) of a program utilizing only the functions CHN_Assign, CHN_Invoke, CHN_Address, CHN_Value, and CHN_Mvalue is easily shown, even though Charon makes no assumptions about lock-step execution or other synchronization features of the runtime system, and does not pose any restrictions on data dependencies in the program. Each invocation of CHN_Assign or CHN_Invoke requires the cooperation of the processors that owns referenced remote data elements. Because all processors execute the same code, any update of such referenced remote data occurring logically before the new values are requested must already have taken place before the request is registered and satisfied; synchronization is performed automatically. This is equivalent to realigning the replicated program counter.

4.4 Detailed behavior of global access functions

A side effect of the cooperative nature of implicitly invoked communications is that they must be issued as broadcast operations on all the processors in the communicator. A processor executing the CHN_Value function must take active part in sending data, but cannot know the recipient until my_address has been evaluated. Both my_address and the expression involving CHN_Value are arguments of the CHN_Assign function, and Fortran nor C semantics prescribe a unique evaluation order. Hence, the rvalue may be evaluated before the destination address is known, which implies that the rvalue be available to *all* processors in the communicator. A broadcast is required.

If the lvalue is not a distributed variable—in other words, if it is a global variable—the CHN_Address and CHN_Assign functions do not have to be used, but it is still legal to do so. In either case, all processors in the communicator obviously need to be able to assemble the complete right hand side, and hence need to receive all remote data. This again highlights the need for broadcasts. Global variables are automatically self-consistent, because each processor assigns the same value to its local copy.

One complication arises when assigning values to global variables through CHN_Assign. The type of the value to be assigned is ordinarily inferred from the data type of the distribution in the left hand side, that is, through a call to CHN_Address. But if the programmer simply uses (the address of) a global variable as the left hand side, CHN_Assign cannot know how to interpret the value of the right hand side, and may do the wrong thing. In fact, it will assume that the left hand side is of the same type as that of the last left hand side that did appear in a CHN_Address function. If this is not correct, the user needs to call CHN_Set_assign_data_type to specify the proper data type explicitly. The alternative would be to implement a different assign function for each data type.

```
int CHN_Set_assign_data_type(MPI_Datatype data_type)
CHN_SET_ASSIGN_DATA_TYPE(int data_type)
   integer data_type
```

IN	data_type	data type implied for CHN_Assign

4.5 Distributed execution examples

Here we give a small sample of the many possibilities of using Charon to create distributed programs. By design, the functionality of all the code fragments is independent of the type of domain decomposition used.

4.5.1 Index swap

Assume we have two distibutions, T_before_ and T_after_, which differ only in their placement of tensor rank indices (p,q). If in the serial program we would want to initialize one with the other, we could write:

```
for (k=0; k<nk; k++) for (j=0; j<nj; j++) for (i=0; i<ni; i++)
for (q=0; q<nq; q++) for (p=0; p<np; p++)
```

```
Tbefore[k][j][i][q][p] = Tafter[q][p][k][j][i];
```

Using Charon, the loop headings stay the same, but the body is replaced by

```
CHN_Assign(CHN_Address(Tbefore_,p,q,i,j,k),
           CHN_Float_value(Tafter_,i,j,k,p,q));
```

This code fragment shows that if a multi-dimensional array in C is actually programmed using multiple indices—a somewhat unusual situation for practical codes—then the order in which its indices appear in `CHN_Address` and `CHN_Value` is the reverse of that in its C declaration. This is because Charon always lists indices in order of increasing array stride (the Fortran storage format), while for C it is the other way around. Here we have assumed that the whole multi-dimensional array has been allocated as a single, contiguous block of data, since this is Charon's storage model.

If this is not the case—that is, if the multi-dimensional C array has been allocated in stages—then there is generally no immediate connection between the Charon memory lay-out and that of the original serial application program, and it is impossible to construct a Charon distribution whose data elements exactly coincide with those in the original code. This is not a problem per se, but it requires that all assignments be replaced by calls to `CHN_Assign` at the same time to avoid inconsistencies between memory models.

More often, however, elements of multi-dimensional arrays in C are referenced through one-dimensional arrays and index macros. For example, the above serial code could also have been written as follows, which would translate into the same Charon code as before. In this case, no index reversal occurs.

```
#define Tafter(i,j,k,p,q)  Tafter[(((((q)*np+(p))*nk+(k))*nj+(j))*ni+(i))]
#define Tbefore(p,q,i,j,k) Tbefore[(((((k)*nj+(j))*ni+(i))*nq+(q))*np+(p))
for (k=0; k<nk; k++) for (j=0; j<nj; j++) for (i=0; i<ni; i++)
for (q=0; q<nq; q++) for (p=0; p<np; p++)
  Tafter(i,j,k,p,q) = Tbefore(p,q,i,j,k);
```

Because of the pluriformity and fluidity of the C memory model, as demonstrated here, we attach the Charon memory model to the more settled Fortran model, which is used in many C codes as well.

Finally, we note that if the only goal is to swap spatial and tensor indices of an entire distribution, then the function `CHN_Redistribute` (see Section 6.1.2) would probably be more appropriate, and significantly faster.

4.5.2 Block-tridagonal solver

In this example a set of independent, block-tridiagonal systems of linear equations is solved on a 2D grid. There is one system for each grid line in the first coordinate direction. Here we only show the forward elimination phase. Each of the diagonals `low`, `main`, and `up` consists of (4×4)-blocks. They are defined as distributions with the tensor indices positioned first, which means the blocks are contiguous in memory. The right hand side vector `r`, which will be overwritten by the solution, consists of (4×1)-blocks. Due to the nature of the tri-diagonal matrix solutions, there is a recurrence of depth one in the first spatial coordinate of the distribution. The serial code might be written as follows.

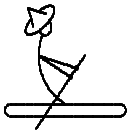
```
  do    j = 1, nj
    do    i = 1, ni-1
      call invert_overwrite(up(1,1,i,j),main(1,1,i,j))
      call invert_overwrite(r(1,1,i,j),main(1,1,i,j))
      call multiply_add(main(1,1,i+1,j),low(1,1,i+1,j),up(1,1,i,j))
      call vmultiply_add(r(1,i+1,j),low(1,1,i+1,j),r(1,i,j))
    end do
  end do
  ...

  subroutine invert_overwrite(mat1,mat2)
  real    mat1(4,4), mat2(4,4), temp(4,4), pivot
c code that overwrites mat1 by mat2^-1*mat1
  pivot = 1.0/mat1(1,1)
  ...
  return
  end

  subroutine multiply_add(mat1,mat2,mat3)
  real    mat1(4,4), mat2(4,4), mat3(4,4)
  integer n, m, k
c code that overwrites mat1 by mat1-mat2*mat3
  do    n = 1, 4
    do    m = 1, 4
      do    k = 1, 4
         mat1(n,m) = mat1(n,m)-mat2(n,k)*mat3(k,m)
      end do
    end do
  end do
  return
  end

  subroutine vmultiply_add(vec1,mat2,vec3)
  real    mat1(4,4), mat2(4,4), mat3(4,4)
  integer n, k
c code that overwrites vec1 by vec1-mat2*vec3
  do    n = 1, 4
    do    k = 1, 4
       vec1(n) = vec1(n)-mat2(n,k)*vec3(k)
    end do
  end do
  return
  end
```

The calling program can be translated using Charon while keeping `invert_overwrite` and (v)`multiply_add` unchanged. It is important to recognize that if in the underlying Charon section the first coordinate direction is divided, then some of the second and third invoke statements in the loop body will be getting data from different cells, and potentially from different processors.

This is transparent to the user. While the program *text* has been augmented significantly, the program *structure* is completely unchanged.

```
 do    j = 1, nj
   do    i = 1, ni-1
     call CHN_INVOKE(invert_overwrite,CHN_ADDRESS(up_,1,1,i,j),1,
$                    CHN_MVALUE(16,main_,1,1,i,j),ierr)
     call CHN_INVOKE(invert_overwrite,CHN_ADDRESS(r_,1,1,i,j),1,
$                    CHN_MVALUE(16,main_,1,1,i,j),ierr)
     call CHN_INVOKE(multiply_add,CHN_ADDRESS(main_,1,1,i+1,j),2,
$                    CHN_MVALUE(16,low_,1,1,i+1,j),
                     CHN_MVALUE(16,up_,1,1,i,j),ierr)
     call CHN_INVOKE(vmultiply_add,CHN_ADDRESS(r_,1,i+1,j),2,
$                    CHN_MVALUE(16,low_,1,1,i+1,j),
                     CHN_MVALUE(4,r_,1,i,j),ierr)
   end do
 end do
```

Chapter 5



Parallel execution support tools

Strict adherence to serial program logic makes the step from non-distributed to distributed execution easy, using Charon utilities. However, the synchronization that occurs whenever distributed variable elements are requested through `CHN_Value` and `CHN_Mvalue` also guarantees that no true concurrency is possible, and parallel speed-up cannot be obtained. The broadcast operations invoked by these functions constitute the mechanism that enforces the principle of the simulated single, replicated program counter to mimic serial program execution. Hence, this coding style should be viewed only as a stepping stone on the way to an efficient, fully concurrent program.

This goal is reached through a succession of assertions by the programmer about the safety of relaxing certain of Charon's rules, and by making explicit those communications that otherwise would be invoked implicitly. In the process, more and more of the domain decomposition is exposed in the program. As the level of abstraction through encapsulation decreases, concurrency and performance go up.

5.1 Supressing broadcasts

The `CHN_Begin_local` function is used to inform Charon that broadcasts should be suppressed within `CHN_Value` and `CHN_Mvalue`, making their operation modes local and non-collective. This opens the door to optimizations, since the program counters on different processors are now independent; they are allowed to 'drift' with respect to each other. `CHN_Begin_local` is an assertion on the part of the programmer that it is safe to ignore remote-data requests. With it comes the responsibility to make sure that any remote data needs have already been met, and that no arithmetic exceptions will occur.

When a request is made for elements of a distributed variable while suppressing broadcasts, Charon will serve the correct values if they are local. But the return values are undefined if the requested values are not local. Hence, allowing all processors to execute

```
call CHN_ASSIGN(CHN_ADDRESS(a_,i,j),CHN_REAL_VALUE(b_,i,j),ierr)
```

indiscriminately to copy one distributed array to another identically distributed array will not result in an error. But the slight variation

```
call CHN_ASSIGN(CHN_ADDRESS(a_,i,j),2.0*CHN_REAL_VALUE(b_,i,j),ierr)
```

may create an arithmetic exception, because `CHN_REAL_VALUE(b_,i,j)` is undefined whenever point (i,j) is not owned by the calling processor and broadcasts are suppressed, and multiplying an undefined value by two may cause overflow. Consequently, the programmer should ordinarily make sure that no processor ever attempts to use a remote data item.

In many cases this can be done by testing for the ownership of points, through the function `CHN_Point_owner` (see Section 3.3). The above example can be safely coded as follows for *any* distributions `a_` and `b_`, provided they are based on the same decomposition (`my_dcmp`), in turn based, through the Charon grid variable, on MPI communicator `my_comm`.

```
  call MPI_COMM_RANK(my_comm, my_rank, ierr)
  call CHN_BEGIN_LOCAL(my_comm,ierr)
  do  j = 1, nj
    do  i = 1, ni
      if (CHN_POINT_OWNER(my_dcmp,i,j) .eq. my_rank)
 $      call CHN_ASSIGN(CHN_ADDRESS(a_,i,j),
 $                      2.0*CHN_REAL_VALUE(b_,i,j),ierr)
    end do
  end do
  call CHN_END_LOCAL(my_comm,ierr)
```

We note that `CHN_Begin_local` takes as an argument a communicator. All processors in its communication domain henceforth suppress broadcasts. Broadcasts resume—and the drifting program counters are effectively resynchronized—by placement of the matching `CHN_End_local`. If broadcasts are not suppressed and the programmer mistakenly allows some processors in the communicator to skip a call involving `CHN_Value` or `CHN_Mvalue`, the program will deadlock, because the broadcast cannot complete.

```
int CHN_Begin_local(MPI_Comm comm)
CHN_BEGIN_LOCAL(comm, ierr)
   integer comm, ierr

int CHN_End_local(MPI_Comm comm)
CHN_END_LOCAL(comm, ierr)
   integer comm, ierr
```

IN	comm	handle to MPI communicator

5.2 Relaxing owner-assigns/serves rules

Suppressing broadcast operations is an important step on the way to an efficient concurrent program, but it is generally not sufficient in case of stencil operations on distributions. Imagine the common nine-point-star stencil on a 2D discretization grid, depicted in Figure 5.1a. Evaluation

of the contributions of neighboring grid points to the stencil operation must inevitably lead to a situation where some grid points are in one cell of the decomposition, and others in another. Hence, if broadcasts are not suppressed and a `CHN_Assign` statement is used to implement the stencil operations, deadlock will ensue if only the processor owning the center point of the stencil executes it. But if broadcasts are suppressed, some of the contributions to the right hand side value will be undefined.

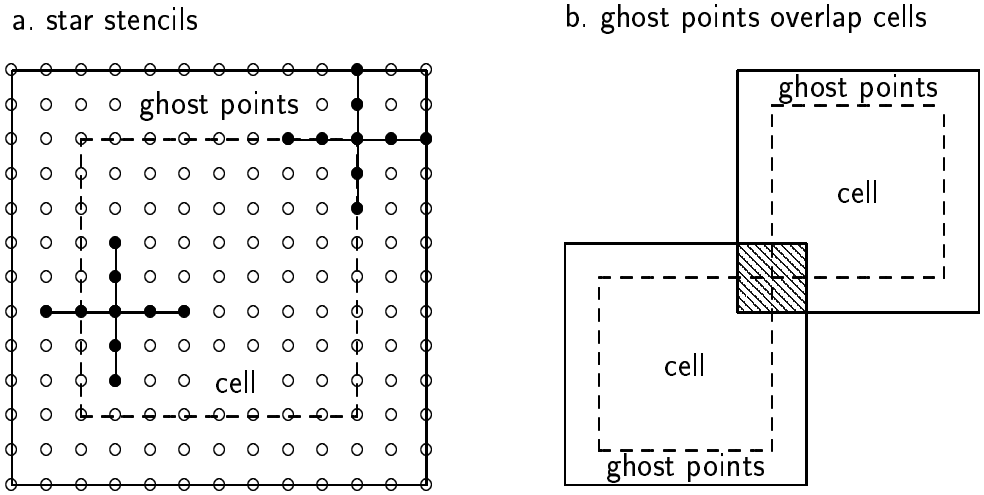


Figure 5.1: Star difference stencil on 2D grid uses ghost points

The solution is to make sure that remote data is fetched in advance, and stored in the location where the stencil operation 'expects' it, namely in a border of buffer or ghost points around each cell, as indicated in Figure 5.1a. These ghost points act as a cache for remote data. Section 6.1.1 explains how the cache is filled. Subsequently, Charon has to be informed to relax the owner-serves rule and reach into the cache for remote data, instead of requesting information from other processors. This is performed by function `CHN_Begin_ghost_access`.

Now a certain ambiguity arises, because one processor may own multiple cells within the decomposition (see Figure 5.1b). Ghost points of one cell may coincide with ghost points or interior grid points of another cell owned by the same processor. The ambiguity is removed by specifying which cell's ghost points may be accessed. When ghost point access is no longer required or desired, or needs to be transferred to another cell, the programmer calls `CHN_End_ghost_access`.

```
int CHN_Begin_ghost_access(int decomposition, int cell_num)
CHN_BEGIN_GHOST_ACCESS(decomposition, cell_num, ierr)
   integer decomposition, cell_num, ierr

int CHN_End_ghost_access(int decomposition, int cell_num)
CHN_END_GHOST_ACCESS(decomposition, cell_num, ierr)
   integer decomposition, cell_num, ierr
```

INOUT	decomposition	handle to decomposition data structure
IN	cell_num	global sequence number of cell within decomposition

Often, the programmer wishes not only to reach into the ghost point cache to fetch remote data, but to store data there as well. Write access to the ghost points through CHN_Address is also provided by CHN_Begin_ghost_access, and is revoked similarly. The combined effect of the access functions is to enlarge temporarily the active cell in the decomposition by the border of ghost points.

It should be stressed that all Charon calls are executable statements, not directives; the exact location of the CHN_Begin/End_local and CHN_Begin/End_ghost_access calls is immaterial and they do not need to be properly nested, as long as Charon is in the proper state once a function requiring a broadcast or ghost point data is encountered.

Finally, we note that the use of CHN_Begin/End_ghost_access is independent of the use of CHN_Begin/End_local. This allows the programmer to test the correctness of ghost point values before giving up the convenience of the serial logic of the code.

5.3 Parallel execution examples

Here we demonstrate how a segment of code, through successive tuning, is tranformed from completely serial to completely concurrent execution. The code represents the application of a seven-point-star stencil operator on all interior points of a 3D grid of an arbitrarily distributed variable a_, based on array a. The result is stored in the variable r_ (array r), whose distribution is based on the same decomposition my_dcmp. Grid point indices all start with 1. Grid sizes are stored in array nsize. Distribution a_ has one ghost point, while r_ has none.

```
do  k = 2, nsize(2)-1
  do  j = 2, nsize(1)-1
    do  i = 2, nsize(0)-1
      call CHN_ASSIGN(CHN_ADDRESS(r_,i,j,k),-6.0*CHN_REAL_VALUE(a_,i,j,k)+
$                     CHN_REAL_VALUE(a_,i+1,j,k)+CHN_REAL_VALUE(a_i-1,j,k)+
$                     CHN_REAL_VALUE(a_,i,j+1,k)+CHN_REAL_VALUE(a_i,j-1,k)+
$                     CHN_REAL_VALUE(a_,i,j,k+1)+CHN_REAL_VALUE(a_i,j,k-1),
$                     ierr)
    end do
  end do
end do
```

This piece of code is an exact reflection of the serial code, even though nothing is known about the particular decomposition used. Performance will be dismal, because of the many broadcasts and function calls. The following fragment removes all broadcasts, and replaces them with a single call to the Charon communication function CHN_Copy_faces_all, to be described in Chapter 6. CHN_Copy_faces_all fills the ghost point buffer with useful data from adjacent cells (usually owned by different processors).

```
call CHN_COPY_FACES_ALL(a_,CHN_NONPERIODIC,1,CHN_STAR,ierr)
call MPI_COMM_RANK(my_comm,my_rank,ierr)
call CHN_BEGIN_LOCAL(my_comm,ierr)

do  c = 0, CHN_TOTAL_NUM_CELLS(my_dcmp)-1
```

```
    if (CHN_CELL_OWNER(my_dcmp,c) .eq. my_rank) then
      call CHN_BEGIN_GHOST_ACCESS(my_dcmp,c,ierr)
      do  k = max(2,CHN_CELL_START_INDEX(my_dcmp,c,2)),
     $          min(CHN_CELL_END_INDEX(my_dcmp,c,2),nsize(2)-1)
        do  j = max(2,CHN_CELL_START_INDEX(my_dcmp,c,1)),
     $            min(CHN_CELL_END_INDEX(my_dcmp,c,1),nsize(1)-1)
          do  i = max(2,CHN_CELL_START_INDEX(my_dcmp,c,0)),
     $              min(CHN_CELL_END_INDEX(my_dcmp,c,0),nsize(0)-1)
            call CHN_ASSIGN(CHN_ADDRESS(r_,i,j,k),-6.0*CHN_REAL_VALUE(a_,i,j,k)+
     $                      CHN_REAL_VALUE(a_,i+1,j,k)+CHN_REAL_VALUE(a_i-1,j,k)+
     $                      CHN_REAL_VALUE(a_,i,j+1,k)+CHN_REAL_VALUE(a_i,j-1,k)+
     $                      CHN_REAL_VALUE(a_,i,j,k+1)+CHN_REAL_VALUE(a_i,j,k-1),
     $                      ierr)
          end do
        end do
      end do
      call CHN_END_GHOST_ACCESS(my_dcmp,c,ierr)
    end if
  end do

  call CHN_END_LOCAL(my_comm,ierr)
```

We note the following. To obtain access to ghost points we need to specify which cell of the decomposition is involved. Hence, it is most natural to replace the loop over all the points in the grid by a loop over all points of all cells of the grid. This automatically avails the programmer of the cell number. It also allows the programmer to test for cell ownership and skip execution of the loop body if appropriate. This saves many point ownership tests and assignment function calls. It also means that the domain decomposition is now somewhat exposed, and the grid point loop bounds become a little more complicated. Nevertheless, the above code fragment is still completely general, and valid for all decompositions. It is also concurrent.

The next optimization removes all assignment function calls. It also visits only those cells actually owned by the calling processor, saving the tests for cell ownership.

```
  integer a_size(0:2), r_size(0:2), start(0:2), end(0:2)
  call CHN_copy_faces_all(a_,CHN_NONPERIODIC,1,CHN_STAR,ierr)

  do  my_c = 0, CHN_TOTAL_NUM_OWNED_CELLS(my_dcmp)-1
    c = CHN_OWN_TO_GLOBAL_CELL_INDEX(my_dcmp,my_c)
    a_offset = CHN_ARRAY_OFFSET(a_,my_c)
    r_offset = CHN_ARRAY_OFFSET(r_,my_c)
    do  dir = 0, 2
      a_size(dir)  = CHN_ARRAY_SIZE(a_,my_c,dir)
      r_size(dir)  = CHN_ARRAY_SIZE(r_,my_c,dir)
      start(dir) = max(2,CHN_CELL_START_INDEX(my_dcmp,c,dir))+1-
     $             CHN_CELL_START_INDEX(my_dcmp,c,dir)
      end(dir)   = min(nsize(dir)-1,CHN_CELL_END_INDEX(my_dcmp,c,dir))+1-
     $             CHN_CELL_START_INDEX(my_dcmp,c,dir)
```

```
    end do
    call loop_body(a(a_offset),r(r_offset),a_size,r_size,start,end)
  end do


  subroutine loop_body(a,r,a_size,r_size,start,end)
  integer a_size(0:2), r_size(0:2), start(0:2), end(0:2), i, j, k
  real a(0:a_size(0)-1,0:a_size(1)-1,0:a_size(2)-1),
 $     r(r_size(0),r_size(1),r_size(2))

  do  k = start(2), end(2)
    do  j = start(1), end(1)
      do  i = start(0), end(0)
        r(i,j,k) = -6.0*a(i,j,k)+a(i+1,j,k)+a(i-1,j,k)+
 $                               a(i,j+1,k)+a(i,j-1,k)+
 $                               a(i,j,k+1)+a(i,j,k-1)
      end do
    end do
  end do

  return
  end
```

It is no longer necessary to suppress broadcasts, nor to explicitly allow access to ghost point values, because there are no more calls to `CHN_Address` or `CHN_Value`. Note the use of the subroutine interface to redimension arrays `a` and `r`, which were assumed one-dimensional storage regions (using arrays to serve to dimension other arrays is not strictly legal Fortran 77, although many compilers allow it. It is allowed in Fortran 90). Often, however, it will be possible to dimension arrays upfront, when more knowledge about the distribution is available.

For example, if the default Charon storage model for distributions is used, and/or if it is known from the outset that each processor owns exactly one cell (uni-partition), simplifications can be made. Below is the final, optimal version of the code for the standard storage model (each cell sub-array has the same dimensions).

```
  dimension a(0:c_size0+1,0:c_size1+1,0:c_size2+1,0:MAXCELLS-1),
 $          r(c_size0,c_size1,c_size2,0:MAXCELLS-1)

  call CHN_COPY_FACES_ALL(a_,CHN_NONPERIODIC,1,CHN_STAR,ierr)

  do  my_c = 0, CHN_TOTAL_NUM_OWNED_CELLS(my_dcmp)-1
    c = CHN_OWN_TO_GLOBAL_CELL_INDEX(my_dcmp,my_c)
    do  kk = max(2,CHN_CELL_START_INDEX(my_dcmp,c,2)),
 $            min(CHN_CELL_END_INDEX(my_dcmp,c,2),nsize(2)-1)
      k = kk-CHN_CELL_START_INDEX(my_dcmp,c,2)+1
      do  jj = max(2,CHN_CELL_START_INDEX(my_dcmp,c,1)),
 $              min(CHN_CELL_END_INDEX(my_dcmp,c,1),nsize(1)-1)
        j = jj-CHN_CELL_START_INDEX(my_dcmp,c,1)+1
```

```
        do  ii = max(2,CHN_CELL_START_INDEX(my_dcmp,c,0)),
 $               min(CHN_CELL_END_INDEX(my_dcmp,c,0),nsize(0)-1)
          i = ii-CHN_CELL_START_INDEX(my_dcmp,c,0)+1

          r(i,j,k,my_c) = -6.0*a(i,j,k,my_c)+
 $                            a(i+1,j,k,my_c)+a(i-1,j,k,my_c)+
 $                            a(i,j+1,k,my_c)+a(i,j-1,k,my_c)+
 $                            a(i,j,k+1,my_c)+a(i,j,k-1,my_c)
        end do
      end do
    end do
  end do
```

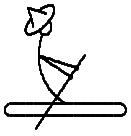
Note the use of the auxiliary coordinates `ii`, `jj` and `kk`, which act as global grid indices. They are particularly useful if tests have to performed on global coordinates, as is the case when determining which are interior points.

We have assumed that the programmer has computed the maximum cell sizes in all coordinate directions and stored these in `c_size0`, `c_size1`, and `c_size2`, respectively. We also assume that the number of cells for each processor does not exceed `MAXCELLS`. It the number of cells is exactly one (uni-partition), the code can be simplified even further.

The final version of the code is very similar to what would have been obtained in any hand-coded message passing program, and it would appear that not much has been gained by using Charon. But there are four main differences. First, the programmer can use the bookkeeping functions provided by Charon, knowing that they are fully debugged. Second, the library function `CHN_Copy_faces_all` can be invoked to fill ghost point values, again with the assurance that it will always yield the correct result. Third, the efficient concurrent code has been developed in a stepwise fashion, maintaining correctness and consistency with the serial code at all times. Fourth, the optimized code segment can be run in conjunction with other totally unoptimized (and even nondistributed) code segments, allowing for a piecemeal program conversion.

Another, less conspicuous, advantage of using Charon is that it is relatively easy to change to another decomposition, especially in the early phases of tuning, when not much explicit knowledge regarding the decomposisiton has been used. Naturally, the more intrepid programmer may skip some parts of the tuning procedure, once some confidence has been built regarding the exact operation of Charon functions.

Chapter 6



Communications

Charon contains two different types of communications, in addition to those already available through MPI. The first type, called *structured* communications, transfers data between locations accessible through Charon's regular value and address functions. In other words, the data is transferred between grid points on the local and remote processors.

The second type, unsurprisingly called *unstructured* communications, transfers data between grid locations and 'private' buffers unrelated to any global grid.

6.1 Structured communications

A data parallel computation can be loosely defined as a global set of tasks, without explicit specification of which processor is responsible for what subset of tasks. The set of tasks comprises a pool of data on which essentially independent operations are performed. Assignment of the subtasks is determined by the subset of the data owned by each processor.

We can similarly define the seemingly oxymoronic data parallel communications as follows. The communication operation consists of a pool of independent data transfers, without explicit specification of which processors are responsible for what subset of the transfers. Assignment of the transfer subtasks is determined by the subsets of the source and destination data owned by each processor or set of processors.

Charon's structured communications fit the definition of data parallel communications. The definition makes sense, because structured communications can be specified completely in terms of grid points, without ever making reference to which processor owns what part of the data to be transferred.

6.1.1 Copying between neighboring cells

Stencil operations often require data from neighboring cells in the decomposition. Instead of fetching these through implicitly invoked communications every time a nearby distribution element

is referenced, the programmer may transfer a number of such elements explicitly in bulk before they are needed, and store them in a border of ghost points around the cell, which acts as a cache for remote data.

For this purpose Charon provides the function CHN_Copy_faces (page 65), and the useful variation CHN_Copy_faces_all (page 67). In CHN_Copy_faces the user specifies a distribution, a coordinate direction, and a particular cut (in that coordinate direction) of the underlying section across which distribution values at grid points are copied. If all cuts are to be selected simultaneously, the value CHN_ALL is specified.

Additionally, the user indicates the thickness (not to exceed the number of ghost points) of the layer of points involved in the copy operation, and a subset of tensor components of the distribution called a tensor mask. For a detailed description of tensor masks, see Section 6.3. If all tensor components are to be copied (as is the case in almost all examples in this manual), set mask equal to CHN_ALL.

The programmer also specifies the Cartesian product of grid points (a 'panel') along the cut to which copying is to be limited. The panel is defined in terms of the global grid, by specifying its beginning and end points, as indicated for a 3D grid in Figure 6.1. In general, on an n-dimensional grid, $n-1$ coordinates are required to specify each of the vectors panel_start and panel_end. One coordinate, that which corresponds to the coordinate direction of the cut, is omitted, since it is implied by the actual cut value. In Figure 6.1 cut number one in coordinate direction one is selected. Hence, the panel vectors only contain the starting and ending values for coordinate directions zero and two, respectively. If all grid points in a certain coordinate direction are to be included in a panel, the user may select CHN_ALL for the corresponding entry in either panel_start or panel_end.

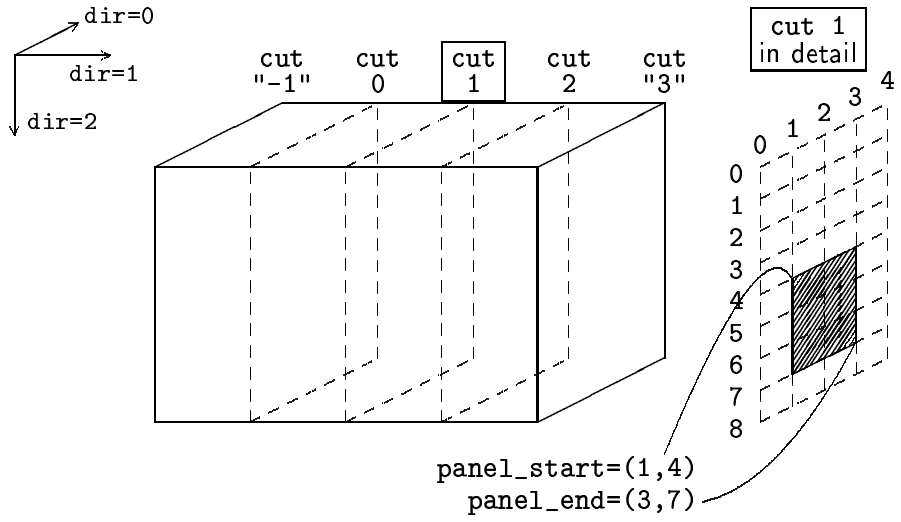


Figure 6.1: Specification of cut and panel for CHN_Copy_faces

The side parameter is used to determine whether copying should take place in the positive coordinate direction specified (CHN_RIGHT), the negative direction (CHN_LEFT), or both (CHN_BOTH_SIDES). Finally, the periodicity parameter specifies whether to allow copying from one side of the grid to the other, corresponding to cuts numbered -1 or CHN_Num_cuts(section,dir). If CHN_NONPERIODIC, such copy requests are ignored, if CHN_PERIODIC, they are honored.

Sometimes it is convenient when performing a periodic copy operation to write some or all of the copied values not into ghost points, but into points properly owned by the cell(s) on the other side of the grid. This is accomplished by specifying CHN_PERIODIC_TRUNCATED, instead of just CHN_PERIODIC. The effect is precisely as if the outer border of the grid proper has turned into a layer of ghost points (of the thickness of that specified in the copy operation). Hence, both source and destination locations of copy values change with respect to plain CHN_PERIODIC copying. Because no ghost points outside the perimeter of the grid are required, truncated periodic copying can be carried out even for distributions that are defined without ghost points, provided that the copy operation is restricted to the boundaries of the grid. More information on the use of truncated periodic copying is given in the example of Section 10.4.

The syntax for the face copy function is as follows.

```
int CHN_Copy_faces(int distribution, int periodicity, int thickness, int mask,
                int dir, int side, int cut_num, int *panel_start, int *panel_end)
CHN_COPY_FACES(distribution, periodicity, thickness, mask, dir, side, cut_num,
                panel_start, panel_end, ierr)
   integer distribution, periodicity, thickness, mask, dir, side, cut_num,
           panel_start(*), panel_end(*), ierr
```

INOUT	distribution	handle to distribution data structure
IN	periodicity	switch: CHN_NONPERIODIC, CHN_PERIODIC or CHN_PERIODIC_TRUNCATED
IN	thickness	thickness of layer of points to be copied
IN	mask	handle to mask data structure, or CHN_ALL
IN	dir	coordinate direction
IN	side	switch: CHN_LEFT, CHN_RIGHT, or CHN_BOTH_SIDES
IN	cut_num	sequence number of cut, or CHN_ALL
IN	panel_start	starting point of copy panel (vector); set component to CHN_ALL for all points
IN	panel_end	end point of copy panel (vector); set component to CHN_ALL for all points

The copy functions can operate on multiple tensor components of the distribution simultaneously, but because this is difficult to visualize, we present examples for scalar distributions only. In Figure 6.2 on page 66 we show several examples of the use of CHN_Copy_faces applied to a distribution with two ghost points, along with the parameters that produced the result. It is assumed that before the copy operation, the 2D distribution holds useful information only at points properly contained within its cells, whose boundaries are indicated by solid lines. For convenience, all grid points of each cell are initialized with a value unique to that cell. To show the results clearly, an exploded view of the grid is offered, in which ghost point values are depicted after the copy operation, in addition to the original interior cell point values. Ghost point values not filled by the operation are left blank.

The inverse of CHN_Copy_faces, which transfers previously written ghost point values to the points properly owned by neighboring cells, is called CHN_Copy_ghost_faces. Its parameters are the same as those of CHN_Copy_faces, and have identical meaning. Truncated periodic copying (CHN_PERIODIC_TRUNCATED) may be specified for CHN_Copy_ghost_faces as well (see above, and Section 10.4).

```
int CHN_Copy_ghost_faces(int distribution, int periodicity, int thickness, int mask,
                   int dir, int side, int cut_num, int *panel_start, int *panel_end)
```

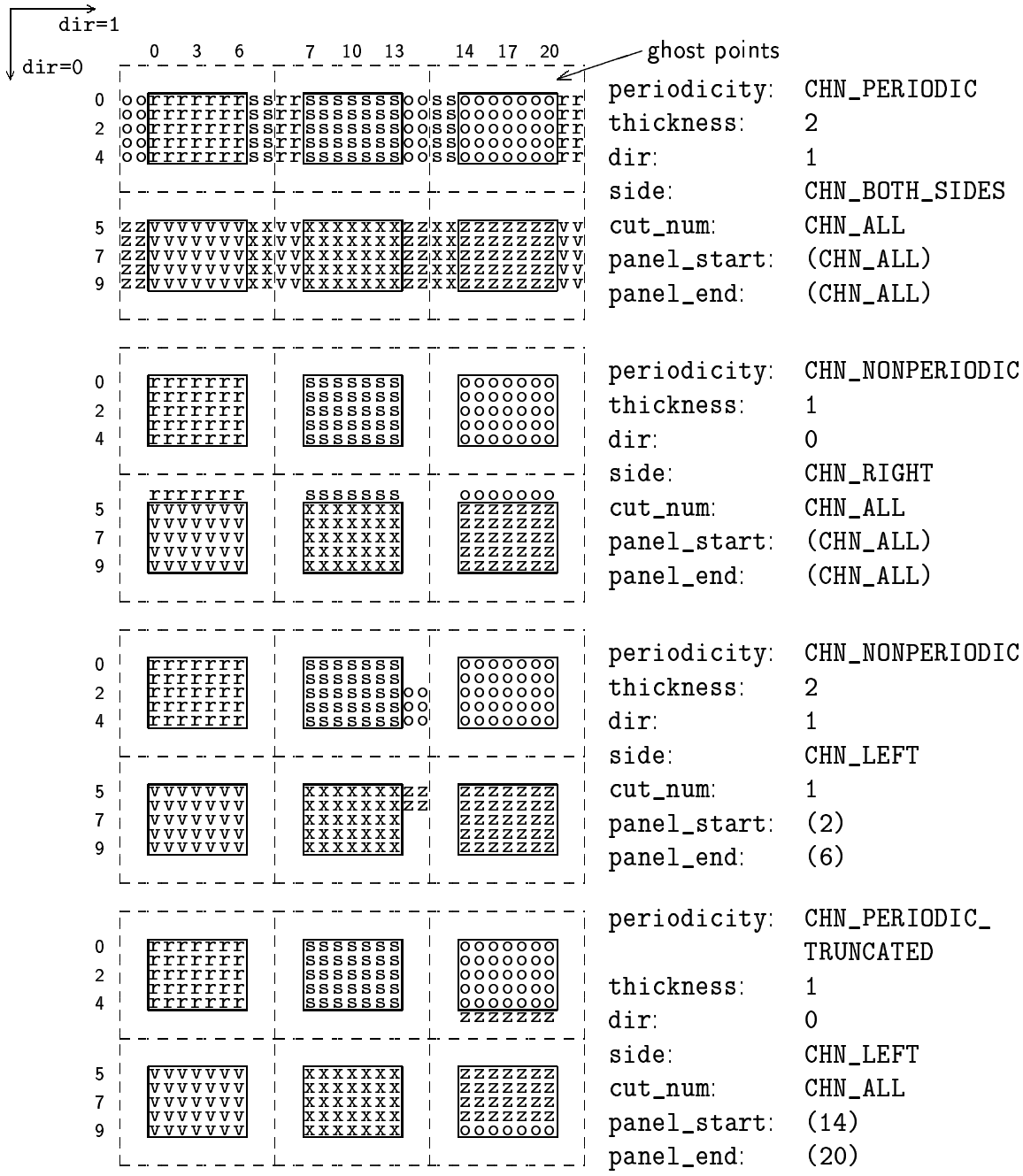


Figure 6.2: Four different uses of CHN_Copy_faces to copy neighboring values of six-cell scalar distribution with two ghost points

```
CHN_COPY_GHOST_FACES(distribution, periodicity, thickness, mask, dir,
                     side, cut_num, panel_start, panel_end, ierr)
   integer distribution, periodicity, thickness, mask, dir, side,
           cut_num, panel_start(*), panel_end(*), ierr
```

INOUT	distribution	handle to distribution data structure
IN	periodicity	switch: CHN_PERIODIC or CHN_NONPERIODIC
IN	thickness	thickness of layer of points to be copied
IN	mask	handle to mask data structure, or CHN_ALL
IN	dir	coordinate direction
IN	side	switch: CHN_LEFT, CHN_RIGHT, or CHN_BOTH_SIDES
IN	cut_num	sequence number of cut, or CHN_ALL
IN	panel_start	starting point of copy panel (vector)
IN	panel_end	end point of copy panel (vector)

CHN_Copy_faces is quite flexible and can be used to implement a wide variety of parallel algorithms involving so-called star shaped stencils (see Figure 6.3a). It is deficient, however, in its support for difference stencils containing cross terms, i.e. stencils containing points that are shifted in more than one coordinate direction with respect to the central point. Examples are the compact 27-point box stencil in 3D, or the 13-point diamond-shaped stencil in 2D, shown in Figures 6.3b and 6.3c, respectively.

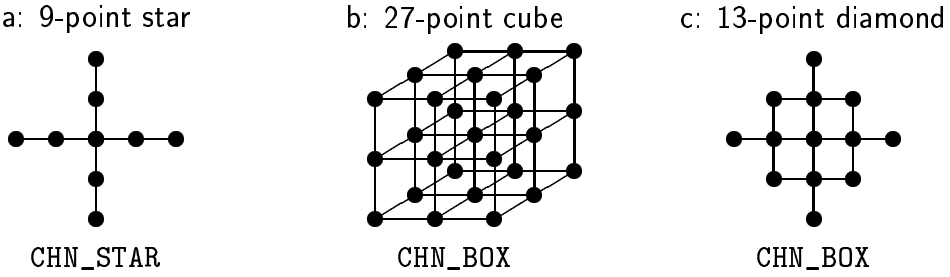


Figure 6.3: Star- and box-shaped difference stencils

Any non-star-shaped stencil, whether symmetrical or asymmetrical, requires diagonal copying of face values to satisfy explicitly remote data needs. For example, to evaluate the 13-point diamond-shaped stencil at the lower right corner of the cell containing the value 'r' in Figure 6.2 would require contributions from the cells holding 's's and 'v's (orthogonal copying) *as well as* from the cell holding 'x's (diagonal copying). For such communication needs Charon provides the function CHN_Copy_faces_all, which copies face values in all coordinate directions (both positive and negative) and along all points on all cuts. The stencil shape parameter determines if the copying is only orthogonal (CHN_STAR), or also diagonal (CHN_BOX). In the latter case, transferring data in the respective coordinate directions occurs in stages to reduce the impact of latency, as explained, for example by Scherr [22].

```
int CHN_Copy_faces_all(int distribution, int periodicity, int thickness, int mask,
                       int shape)
CHN_COPY_FACES_ALL(distribution, periodicity, thickness, mask, shape, ierr)
```

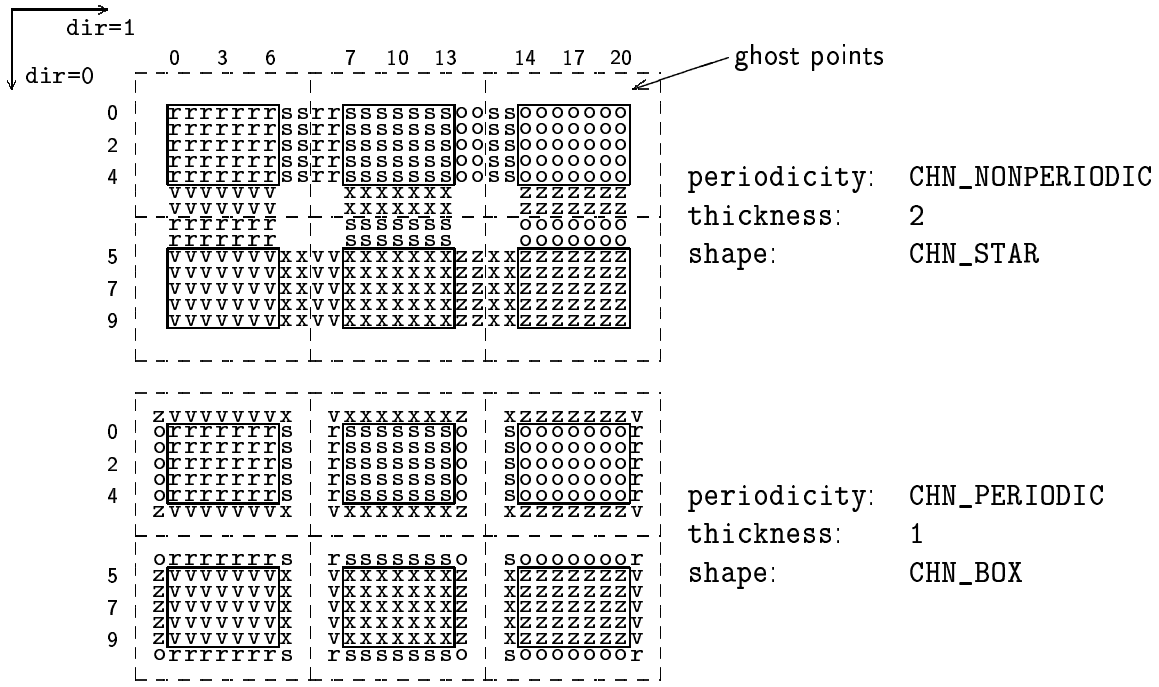


Figure 6.4: Using CHN_Copy_faces_all to fill ghost point values of six-cell scalar distribution

```
integer distribution, periodicity, thickness, mask, shape, ierr
```

INOUT	distribution	handle to distribution data structure
IN	periodicity	switch: CHN_PERIODIC or CHN_NONPERIODIC
IN	thickness	thickness of layer of points to be copied
IN	mask	handle to mask data structure, or CHN_ALL
IN	shape	switch: CHN_STAR or CHN_BOX

In Figure 6.4 we show the result of applying CHN_Copy_faces_all to the scalar distribution of Figure 6.2. Observe that using

```
call CHN_COPY_FACES_ALL(distribution,periodicity,thickness,CHN_ALL,CHN_STAR,ierr)
```

is equivalent to:

```
num_dims = CHN_NUM_DIMS(CHN_GRID(CHN_SECTION(CHN_DECOMPOSITION(distribution))))
do  dir = 1, num_dims-1
  panel_start(dir) = CHN_ALL
  panel_end(dir)   = CHN_ALL
end do
do dir = 0, num_dims-1
  call CHN_COPY_FACES(distribution,periodicity,thickness,CHN_ALL,dir,
$                     CHN_BOTH_SIDES,CHN_ALL,panel_start,panel_end,ierr)
end do
```

Hence, even if diagonal copying is not required, it is often convenient to use CHN_Copy_faces_all if all cells in the grid need to exchange data with their neighbors. There is no

inverse of CHN_Copy_faces_all, that is, function CHN_Copy_ghost_faces_all does not exist. The reason is that multiple ghost points map to the same interior cell points (even in the case of star-shaped difference stencils), which would lead to ambiguity. For the same reason, CHN_PERIODIC_TRUNCATED cannot be specified when copying ghost cell faces.

Cell face copy functions will produce incorrect or ambiguous results if the cells involved do not have enough points to support the copying. For example, if a decomposition is defined such that certain cells have only one point in the second coordinate direction, then a copy operation in that coordinate direction whose thickness parameter is two will fail. If the periodicity parameter CHN_PERIODIC_TRUNCATED is specified, the size restrictions on cells at the boundary become more severe, because the cell size is effectively shrunk by the width of the border of ghost points. It is the user's responsibility to provide valid parameters for the copy operation. The exact conditions (all relating to the coordinate direction in which copying takes place) are as follows, assuming that all cuts are selected for copying (cuts = CHN_ALL). If only one cuts is selected, some of the restrictions might be relaxed.

Let n_{gr} be the size of the grid. When the copying is from ghost points to grid points (CHN_Copy_ghost_faces), the boolean variable *ghost* is true, and false otherwise ($\neg$*ghost*). n_c is the size of any cell, n_{c_b} the size of a cell on the boundary of the grid. The thickness of the layer to be copied is l, and the number of ghost points of the distribution is d.

	periodicity			
	CHN_(NON_)PERIODIC		CHN_PERIODIC_TRUNCATED	
side	$\neg$*ghost*	*ghost*	$\neg$*ghost*	*ghost*
CHN_LEFT/RIGHT	$n_c \geq l$	$n_c \geq l$	$\begin{cases} n_c \geq l \\ n_{c_b} \geq d+1 \\ n_{gr} \geq 2d+l \end{cases}$	$\begin{cases} n_c \geq l \\ n_{c_b} \geq d+l \\ n_{gr} \geq 2d+l \end{cases}$
CHN_BOTH_SIDES	$n_c \geq l$	$n_c \geq 2l$	$\begin{cases} n_c \geq l \\ n_{c_b} \geq d+l \\ n_{gr} \geq 2d+l \end{cases}$	$\begin{cases} n_c \geq 2l \\ n_{c_b} \geq d+2l \\ n_{gr} \geq 2d+2l \end{cases}$

Table 6.1: Minumum cell and grid size requirements for cell face copy functions

All copy functions and other structured and unstructured communications—save redistributions—are nonlocal, and pseudo-collective (see Section 1.10); completion may depend on other processors, which need to call the function in question with the same parameters. But only those processors actively involved in sending and/or receiving data need to call the function. Others can safely skip it. Redistribution is nonlocal and collective, because it relies on MPI nonlocal collective communications

Finally, we observe that Charon's communication functions do not require multiple processors per se. A grid may be divided into many cells, all assigned to the same processor. Charon's communications make sure that stencil operations can be performed on the partitioned single-owner grid. This strategy may increase data locality, which is important for performance on computers with hierarchical memories.

6.1.2 Redistributions

Certain algorithms feature strong but mutually incompatible data dependencies during different parts of the computation. As an example we mention the multi-dimensional Fast Fourier Transform, discussed in Section 3.5.2 on page 40. Such problems can sometimes benefit from a change in the overall data distribution. In keeping with High Performance Fortran terminology [15], we call this operation a 'redistribution'. The Charon function accomplishing it is `CHN_Redistribute`.

Although its principal goal is to establish data rearrangements like the transpositions depicted in Figure 3.7 on page 41, `CHN_Redistribute` can transform any legal Charon distribution into any other compatible distribution. Two distributions are compatible under the following conditions:

1. They are based on the same grid variable.
2. They have the same tensor rank.
3. They do not overlap in memory (this restriction will be removed in a future Charon release).
4. They are of the same data type (e.g. both `MPI_REAL`, or both `MPI_CHAR`).

These rules imply that two distributions are compatible, even if they have different numbers of ghost points, if they are based on different decompositions, or if their tensor indices are in different positions with respect to the spatial indices (i.e. one is defined using `CHN_Set_tensor_indices_first`, and the other using `CHN_Set_tensor_indices_last`). They also may use different amounts of storage space on the calling processor, as long as the programmer guarantees that *enough* space is available.

As a result, redistribution is a very flexible function. Besides the transposition mentioned above, a particularly useful application is the redistribution between an arbitrary multi-owner distribution and one that contains just a single cell (solo-partition), usually without ghost points. If the solo-partition distribution is defined with carefully chosen array dimensions, its memory lay-out can be made to correspond exactly with that of the serial code (see Section 3.4.2). As a consequence, mapping back and forth between multi-owner and single-owner distributions allows the programmer to switch dynamically between serial and distributed (and perhaps parallel) program execution, without having to change the serial part of the code at all. Mapping to the single-owner distribution automatically transfers the distributed data into the locations expected by the serial code, and vice versa.

This feature can be exploited to parallelize only parts of a legacy code, while keeping the rest unchanged. Because of its massive communication volume, redistribution is an expensive operation, whose application should be limited to a minimum. Moreover, Amdahl's law soon hampers parallel speed-up if parts of the code are run in serial mode, so the aim of most programmers will be to remove all mappings to single-owner distributions. But as a parallel code development device this type of redistribution is very useful. It allows the parallelization process to focus on one routine (or even one statement) at a time, without worries about correctness of the rest of the code.

```
int CHN_Redistribute(int distibution1, int distribution2)
CHN_REDISTRIBUTE(distibution1, distribution2, ierr)
```

```
   integer distibution1, distribution2, ierr
```

INOUT	distibution1	handle to output distribution data structure
IN	distribution2	handle to input distribution data structure

6.2 Unstructured communications

Sometimes it will be necessary to specify operations at grid points that depend on data located at non-adjacent points. The most obvious example is that of periodic boundary conditions, but these can often be handled by specifying `CHN_PERIODIC` in `CHN_Copy(_ghost)_faces(_all)`. Other more complicated examples might be the inclusion of some long-range source functions, such as those occurring in numerical models of turbulence, radiation, electric potential, etc. In these cases the programmer can rely on the implicitly invoked communications of `CHN_(M)Value`—at the high cost of many broadcasts and synchronizations.

Another solution is to request explicitly for distributed data to be copied in bulk to a user-specified buffer on a certain processor. This is done through function `CHN_Get_tile`. A convenient way of specifying subsets of global distribution data is to indicate the beginning and ending coordinates of Cartesian subsets of the grid, termed *tiles*. Tiles are similar to the panels, used, for example, in the function `CHN_Copy_faces` (see Section 6.1.1). They are defined by the vectors `tile_start` and `tile_end`. The difference is that tiles are n-dimensional subsets of n-dimensional grids, whereas panels have dimensionality $n-1$. Of course, by selecting the tile to have size one in certain coordinate directions, the user can effect copying lower-dimensional subsets of the distribution.

Often the programmer would like to view the buffer receiving the distribution data as a multi-dimensional array itself, into which the subset of the distribution is 'inserted'. The user interface of `CHN_Get_tile` accommodates this by letting the user specify the starting address of the buffer, the starting and ending grid indices of the multi-dimensional buffer array (vectors `array_start` and `array_end`. and the grid indices of the insertion point (vector `insert_point`). We indicate in Figure 6.5 how these index vectors relate to each other and the grid variable. The tile may span several cells in the decomposition, in which case their respective owners must all participate in the communication.

```
int CHN_Get_tile(int distribution, int mask, int root, void *start_address,
             int *tile_start, int *tile_end, int *insert_point, int *array_start,
             int *array_end)
CHN_GET_TILE(distribution, mask, root, start_address, tile_start, tile_end,
             insert_point, array_start, array_end, ierr)
   integer distribution, mask, root, tile_start(*), tile_end(*), insert_point(*),
 $         array_start(*), array_end(*), ierr
   <type>  start_address(*)
```

IN	distribution	handle to distribution data structure
IN	mask	handle to mask data structure, or `CHN_ALL`
IN	root	rank of the processor receiving data
INOUT	start_address	beginning of local buffer
IN	tile_start	starting grid indices (vector) of tile to be copied
IN	tile_end	ending grid indices (vector) of tile to be copied

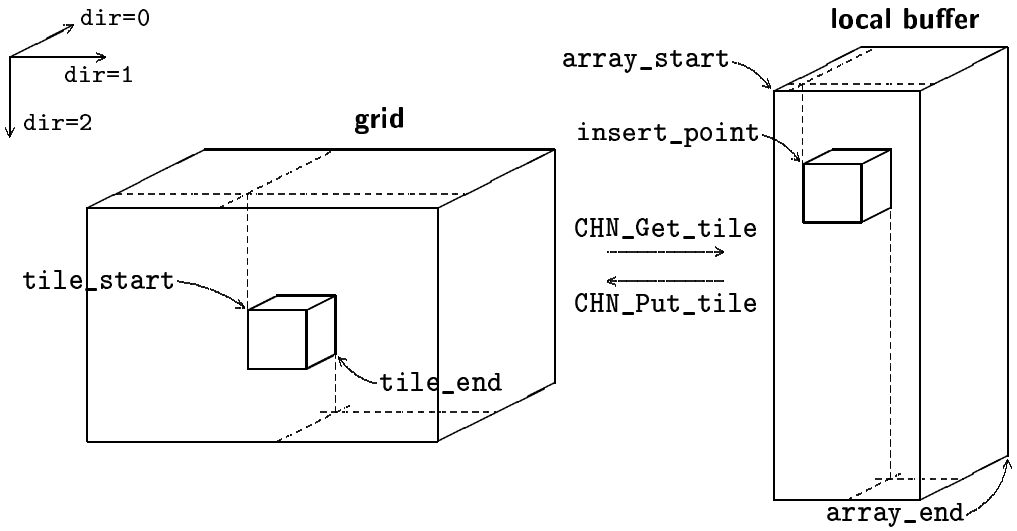


Figure 6.5: Using `CHN_Get/Put_tile` to copy between global distribution and local buffer

IN	`insert_point`	insertion point (vector) into local buffer array
IN	`array_start`	(virtual) starting point (vector) of local buffer array
IN	`array_end`	(virtual) ending point (vector) of local buffer array

There need not be any relationship between the size of the grid and the dimensions that the programmer specifies for interpretation of the local buffer as a multi-dimensional array. As long as the tile fits inside the array, the data transfer completes successfully. We recall from the discussion about panels in Section 6.1.1 that tile, local array, and insertion point are specified in terms of spatial coordinates (i.e. grid indices).

Special attention must be paid to the role of the tensor `mask`. It is used to specify the subset of the tensor components of the distribution that is being gathered onto the `root` processor. Any "holes" in the mask (i.e. tensor components that are not fetched) are eliminated from the tile by compaction. For example, if the distribution is defined as a (3×3) tensor field and the tensor mask only contains the diagonal elements, then the resulting tile is stored as if it were a (3×1) vector field. In other words, the tile always contains contiguous data. Nonzero tensor components in the mask are ordered lexicographically in the compacted tile, while respecting their position before or after the spatial indices. A consequence of the storage strategy for tiles is that the programmer need only reserve as much space as is required for the tensor components that are actually being fetched, not for the holes.

While `CHN_Get_tile` provides a copy function from the global distribution to a local buffer, the reverse is also possible, through `CHN_Put_tile`. This functions 'seeds' a tile within the distribution with data stored in a local buffer (again interpreted as a multi-dimensional array) on a particular processor. Its parameters are the same as those of `CHN_Get_tile`, as illustrated in Figure 6.5, except that the insertion point has become the 'extraction point' from the local array. The tensor mask now defines a scattering from the compacted tile to the appropriate locations in the tensor field of the target distribution.

```
int CHN_Put_tile(int distribution, int mask, int root, void *start_address,
            int *tile_start, int *tile_end, int *extract_point, int *array_start,
```

```
              int *array_end)
CHN_PUT_TILE(distribution, mask, root, start_address, tile_start, tile_end,
              insert_point, array_start, array_end, ierr)
   integer distribution, mask, root, tile_start(*), tile_end(*), extract_point(*),
 $         array_start(*), array_end(*), ierr
   <type>  start_address(*)
```

INOUT	distribution	handle to distribution data structure
IN	mask	handle to tensor mask data structure, or CHN_ALL
IN	root	rank of the processor providing data
IN	start_address	beginning of local buffer
IN	tile_start	starting grid indices (vector) of tile to be filled
IN	tile_end	ending grid indices (vector) of tile to be filled
IN	extract_point	extraction point (vector) from local buffer array
IN	array_start	(virtual) starting point (vector) of local buffer array
IN	array_end	(virtual) ending point (vector) of local buffer array

Finally, Charon also provides collective versions of the above unstructured communications. Instead of transferring a tile from a distribution to a user buffer on a single processor (CHN_Get_tile), we may elect to store this tile in private buffers on all processors in the communicator (CHN_Bcast_tile). In this broadcast operation no recipient needs to be specified. While all processors should make reference to the same tile in the distribution, the dimensions of the private array and the insertion point may be different on each processor.

Similarly, instead of transferring the content of a local buffer on a single processor to a tile in the distribution, all processors in the communicator may conspire to write the tile, through CHN_Reduce_tile. In this case a reduction operation (MPI_MAX, MPI_MIN, MPI_SUM, or MPI_PROD) must be specified, to be applied to the data elements from different processors sent to the same location in the tile. Again, all processors should make reference to the same tile in the distribution, but the dimensions of the private array and the insertion point may be different on each processor.

```
int CHN_Bcast_tile(int distribution, int mask, void *start_address, int *tile_start,
              int *tile_end, int *insert_point, int *array_start,
              int *array_end)
CHN_BCAST_TILE(distribution, mask, start_address, tile_start, tile_end, insert_point,
              array_start, array_end, int ierr)
   integer distribution, mask, tile_start(*), tile_end(*), insert_point(*),
 $         array_start(*), array_end(*), ierr
   <type>  start_address(*)
```

IN	distribution	handle to distribution data structure
IN	mask	handle to tensor mask data structure, or CHN_ALL
INOUT	start_address	beginning of local buffer
IN	tile_start	starting grid indices (vector) of tile to be copied
IN	tile_end	ending grid indices (vector) of tile to be copied
IN	insert_point	insertion point (vector) into local buffer array
IN	array_start	(virtual) starting point (vector) of local buffer array
IN	array_end	(virtual) ending point (vector) of local buffer array

```
int CHN_Reduce_tile(int distribution, int mask, void *start_address, int *tile_start,
                int *tile_end, int *extract_point, int *array_start,
                int *array_end, MPI_Op op)
CHN_REDUCE_TILE(distribution, mask, start_address, tile_start, tile_end, insert_point,
                array_start, array_end, op, ierr)
   integer distribution, mask, tile_start(*), tile_end(*), extract_point(*),
 $         array_start(*), array_end(*), op, ierr
   <type>  start_address(*)
```

INOUT	distribution	handle to distribution data structure
IN	mask	handle to tensor mask data structure, or CHN_ALL
IN	start_address	beginning of local buffer
IN	tile_start	starting grid indices (vector) of tile to be filled
IN	tile_end	ending grid indices (vector) of tile to be filled
IN	extract_point	extraction point (vector) from local buffer array
IN	array_start	(virtual) starting point (vector) of local buffer array
IN	array_end	(virtual) ending point (vector) of local buffer array
IN	op	MPI operation specifying how to reduce co-located data

The functionality of CHN_Get_tile and CHN_Put_tile is comparable to that of the Global Arrays toolkit [20] operations GA_get and GA_put, respectively. There are no Global Arrays equivalents to the collective Charon functions CHN_Reduce_tile and CHN_Bcast_tile.

6.3 Tensor masks

. A special Charon data structure is the so-called tensor mask. It is used to indicate a subset of the tensor components in a distribution to which a particular communication function (CHN_Copy(_ghost)_faces(_all) or CHN_Get/Put/Reduce/Bcast_tile) is to be limited. The mask is initialized using CHN_Create_tensor_mask, and is destroyed by calling CHN_Delete_tensor_mask. Initialization is similar to that of distributions in that the tensor rank and the extents of the tensor indices are specified. Starting values for the indices (usually zero for C and one for Fortran) are those set by CHN_Set_all_tensor_start_indices (See Section 3.4) before creation of the mask, or for a particular existing mask by CHN_Set_all_tensor_start_indices. A tensor mask can be used with any distribution that has the same tensor structure (i.e. same shape (rank and numbers of components) and same starting indices).

```
int CHN_Create_tensor_mask(int *mask, int rank, ...)
CHN_CREATE_TENSOR_MASK(mask, rank, size0, size1, ..., ierr)
   integer mask, rank, size0, size1, ..., ierr
```

OUT	mask	handle to tensor mask data structure
IN	rank	rank of tensor at each grid point
IN	size0	extent of first tensor index
IN	size1	extent of second tensor index
IN	...	extent of subsequent tensor indices

```
int CHN_Delete_tensor_mask(int *mask)
CHN_DELETE_TENSOR_MASK(mask, ierr)
   integer mask, ierr
```

INOUT	mask	handle to tensor mask data structure

Initially the mask is empty, that is, no tensor components are selected for involvement in communications yet. Individual components are added one by one by specifying their tensor indices using CHN_Set_tensor_mask. A selected component can be removed from the mask by calling CHN_Unset_tensor_mask. It is not an error to select a tensor component twice, or to remove an unselected tensor component. It is also not an error to attempt to select or remove a tensor component from the universal tensor mask identified by CHN_ALL, but it is ineffectual.

```
int CHN_Set_tensor_mask(int mask, ...)
CHN_SET_TENSOR_MASK(mask, index0, index1, ..., ierr)
   integer mask, index0, index1, ..., ierr
```

INOUT	mask	handle to tensor mask data structure
IN	index0	first tensor index value
IN	index1	second tensor index value
IN	...	subsequent tensor index values

```
int CHN_Unset_tensor_mask(int mask, ...)
CHN_UNSET_TENSOR_MASK(mask, index0, index1, ..., ierr)
   integer mask, index0, index1, ..., ierr
```

INOUT	mask	handle to tensor mask data structure
IN	index0	first tensor index value
IN	index1	second tensor index value
IN	...	subsequent tensor index values

All tensor mask operations are local and noncollective, but any masks used in a communication operation must be compatible with each other and the distribution on which they act.

6.3.1 Tensor mask query functions

Distribution query functions CHN_Tensor_rank, CHN_Tensor_size, and CHN_Tensor_start_index (Section 3.4.1) can also be applied to tensor masks, with the expected results. One additional query function, CHN_Tensor_mask, returns one if the tensor component has been selected, and zero otherwise.

```
int CHN_Tensor_mask(int mask, ...)
integer function CHN_TENSOR_MASK(mask, index0, index1, ...)
   integer mask, index0, index1, ...
```

Error return value: -1

IN	mask	handle to tensor mask data structure
IN	index0	first tensor index value
IN	index1	second tensor index value
IN	...	subsequent tensor index values

6.4 Communication examples

On structured grids there are countless ways data elements at grid points can interact, and hence countless possible communication examples. We give just two examples here, one for structured and one for unstructured communications.

6.4.1 Alternating Direction Implicit (ADI)

An important aim of Charon is to support parallelization of program structures involving complicated data depedencies. These often defy efficient implementation using more traditional parallelization tools, such as High Performance Fortran (HPF) [15], PARTI [24], etc., which essentially only allow data parallel operations.

Our example is for illustrative purposes only. It is based on the NPB Block Tri-diagonal (BT) problem [1], which defines an ADI (Alternating Direction Implicit) algorithm to solve a system of nonlinear partial differential equations on a 3D grid. We reduce the system to a simple linear, scalar equation on a 2D grid:

$$\frac{\partial q}{\partial t} = \frac{\partial q^2}{\partial x^2} + \frac{\partial q^2}{\partial y^2} \tag{6.1}$$

The interesting part of the ADI algorithm concerns the inversion of sets of banded tri-diagonal matrices (so-called *factors*), since they involve recurrences (i.e. data dependencies) in different coordinate directions. There is one banded matrix for each grid line, in each of the two coordinate directions. Here is the C pseudo code for a single time step of the ADI algorithm.

```
for (each grid point) compute_residual(rhs,q);        /* rhs: residual      */
for (dir=0; dir<2; dir++) invert_factor(lhs,rhs,dir); /* lhs: banded matrix */
for (each grid point) update(q,rhs);                  /* q = q + rhs        */
```

The Charon code used to implement efficiently the residual computation, which is data parallel, has already been described in Section 5.3 for a 3D grid. The following serial code represents the forward-elimination phase of the inversion of the y-factor (second coordinate direction) in our sample ADI program. The grid is assumed to have `ni`×`nj` points.

```
#define lhs(b,i,j)  lhs[(((j)*ni+(i))*3+(b))]
#define rhs(i,j)    rhs[((j)*ni+(i))]
for (j=0; j<nj-1; j++) for (i=0; i<ni; i++) {
  inv           = 1.0/lhs(1,i,j);      /* compute pivot reciprocal */
  lhs(1,i,j)   *= inv;                        /* scale matrix row */
  rhs(i,j)     *= inv;                        /*   "      "    "  */
  lhs(1,j+1,i) -= lhs(2,i,j)*lhs(0,j+1,i);    /* update next row  */
  rhs(j+1,i)   -= rhs(i,j)   *lhs(0,j+1,i);   /*   "      "   "   */
}
```

Here the array `lhs` represents a family of banded, tri-diagonal matrices, one for each grid line in the y-direction (i.e. one for each value of `i`). The first index selects the particular band of the matrix. Index `0` corresponds to the lower, `1` to the main, and `2` to the upper diagonal. We again convert using only top-level Charon functions:

```
for (j=1; j<=nj-1; j++) for (i=1; i<=ni; i++) {
 inv = 1.0/CHN_Float_value(lhs_,1,i,j);
 CHN_Assign(CHN_Address(lhs_,2,i,j),  inv*CHN_Float_value(lhs_,2,i,j));
 CHN_Assign(CHN_Address(rhs_,i,j),    inv*CHN_Float_value(rhs_,i,j));
 CHN_Assign(CHN_Address(lhs_,1,j+1,i),CHN_Float_value(lhs_,1,j+1,i) -
            CHN_Float_value(lhs_,2,i,j)*CHN_Float_value(lhs_,0,j+1,i));
 CHN_Assign(CHN_Address(rhs_,j+1,i),    CHN_Float_value(rhs_,j+1,i)   -
            CHN_Float_value(rhs_,i,j) * CHN_Float_value(lhs_,0,j+1,i));
}
```

Notice again that in the transformed code fragment no influence of the domain decomposition is visible. The situation changes once we start to optimize by making sure no implicitly invoked communications are necessary. Now two approaches are available.

The first is similar to that offered by HPF, namely the `CHN_Redistribute` facility. Before the iterations start, two different pencil decompositions are defined that are aligned with the x- and y-grid lines, respectively. The corresponding distributions are `rhsx_` and `lhsx_`, and `rhsy_` and `lhsy_`, respectively. Switching from x-aligned to y-aligned `lhs` distributions is established by calling `CHN_Redistribute(lhsy_,lhsx_)`. The effect is a transposition of the distributed arrays, as illustrated for a 3D grid in Figure 3.7. Although this method is fairly easy to program, it is rather inefficient because of the large communication volume.

The second approach leaves the data distributions intact. We choose the multi-partition domain decomposition, which has the special property that each processor owns a cell in each row and each column of cells of the grid (see Section 3.3.1). Hence, if the solution process advances by rows or columns of cells in the respective coordinate directions—so as to respect the recurrence relations in these directions—, a perfect load balance ensues.

Early successful results of using multi-partitioning for parallelizing ADI applications, including comparisons of its performance with that of pipelined and transpose-based uni-partitioning methods, are reported in [16, 25]. Some negative experiences have also been reported [19], but these are almost certainly attributable to very small problem sizes, deficiencies in the operating system used to carry out the numerical experiments, and the choice of 2D (instead of 3D) multi-partitioning for 3D applications.

Because of its favorable distribution properties, multi-partitioning was chosen for the implementation of the two synthetic ADI applications SP and BT in the NAS Parallel Benchmarks II suite [2]. We note that none of the systems surveyed in [27] has the flexibility of supporting multi-partitioning, but in Charon it is easily defined.

Before the loop nest is entered, we copy `lhs` and `rhs` values immediately 'ahead' of each column of cells into the ghost point locations. Thus, when the loop body is executed for the last column of points in each column of cells, all its remote data requirements are automatically satisfied. After all `rhs` and `lhs` ghost point values are written for a whole column of cells, the updated values are transferred in bulk to the next column using `CHN_Copy_ghost_faces`. The following program results (see also Section 5.3).

```
int allv[1] = {CHN_ALL};

/* pre-loop communications */
CHN_Copy_faces(rhs_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_LEFT,CHN_ALL,allv,allv);
```

```
CHN_Copy_faces(lhs_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_LEFT,CHN_ALL,allv,allv);
MPI_Comm_rank(my_comm, &my_rank);
CHN_Begin_local(my_comm);

/* jp determines the column of cells */
for (jp=0; jp<CHN_Num_cells(my_dcmp,1); jp++) {
  /* ip determines the cell within the column */
  for (ip=0; ip<CHN_Num_cells(my_dcmp,0); ip++) {
    c = CHN_Cell_index(my_dcmp,ip,jp);
    if (CHN_Cell_owner(my_dcmp,c)== my_rank) {
      CHN_Begin_ghost_access(my_dcmp,c);
      for (j =CHN_Cell_start_index(my_dcmp,1,c);
           j<=min(CHN_Cell_end_index(my_dcmp,1,c),nj-2),; j++)
      for (i =CHN_Cell_start_index(my_dcmp,0,c);
           i<=CHN_Cell_end_index(my_dcmp,0,c); i++) {
        inv = 1.0/CHN_Float_value(lhs_,1,i,j);
        CHN_Assign(CHN_Address(lhs_,2,i,j),  inv*CHN_Float_value(lhs_,2,i,j));
        CHN_Assign(CHN_Address(rhs_,i,j),    inv*CHN_Float_value(rhs_,i,j));
        CHN_Assign(CHN_Address(lhs_,1,j+1,i),  CHN_Float_value(lhs_,1,j+1,i)-
                   CHN_Float_value(lhs_,2,i,j)*CHN_Float_value(lhs_,0,j+1,i));
        CHN_Assign(CHN_Address(rhs_,j+1,i),    CHN_Float_value(rhs_,j+1,i)-
                   CHN_Float_value(rhs_,i,j)*  CHN_Float_value(lhs_,0,j+1,i));
      }
      CHN_End_ghost_access(my_dcmp,c);
    }
  }
  CHN_Copy_ghost_faces(lhs_,CHN_NONPERIODIC,1,CHN_ALL,0,CHN_RIGHT,ip,allv,allv);
  CHN_Copy_ghost_faces(rhs_,CHN_NONPERIODIC,1,CHN_ALL,0,CHN_RIGHT,ip,allv,allv);
}
CHN_End_local(my_comm);
```

A depiction of the first five phases of the algorithm for a four-processor computation is presented in Figure 6.6 on page 79. Cells are indicated by squares, with the pattern inside indicating whether (some of) the array alements corresponding to grid points have only been initialized (diagonal hatching), or have already been updated by computations or communications (horizontal hatching). Only the ghost points of interest to the forward elimination are shown (narrow strips on the right side of the cells). The outer loop of the code is over the columns of cells in the grid, and each phase applies uniformly to all cells in a column. The dashed frames around the cells owned by a particular processor (number 2, in this case) serve to show that the code exhibits a perfectly balanced load. During each phase, all processors have the same amount of computational work to do, or the same amount of data to communicate.

The significance of being able to write into the ghost point locations becomes clear when examining the loop body more carefully. Each group of four assignments changes values in two successive rows of each tri-diagonal matrix. Hence, if we allow the loop over the grid points within each cell to run from the first to the last cell index in the second coordinate direction, the second pair of assignments in the loop body (`lhs(1,i,j+1)=...` and `rhs(i,j+1)=...`) would attempt to store data at points owned by cells in the next column. Allowing those values

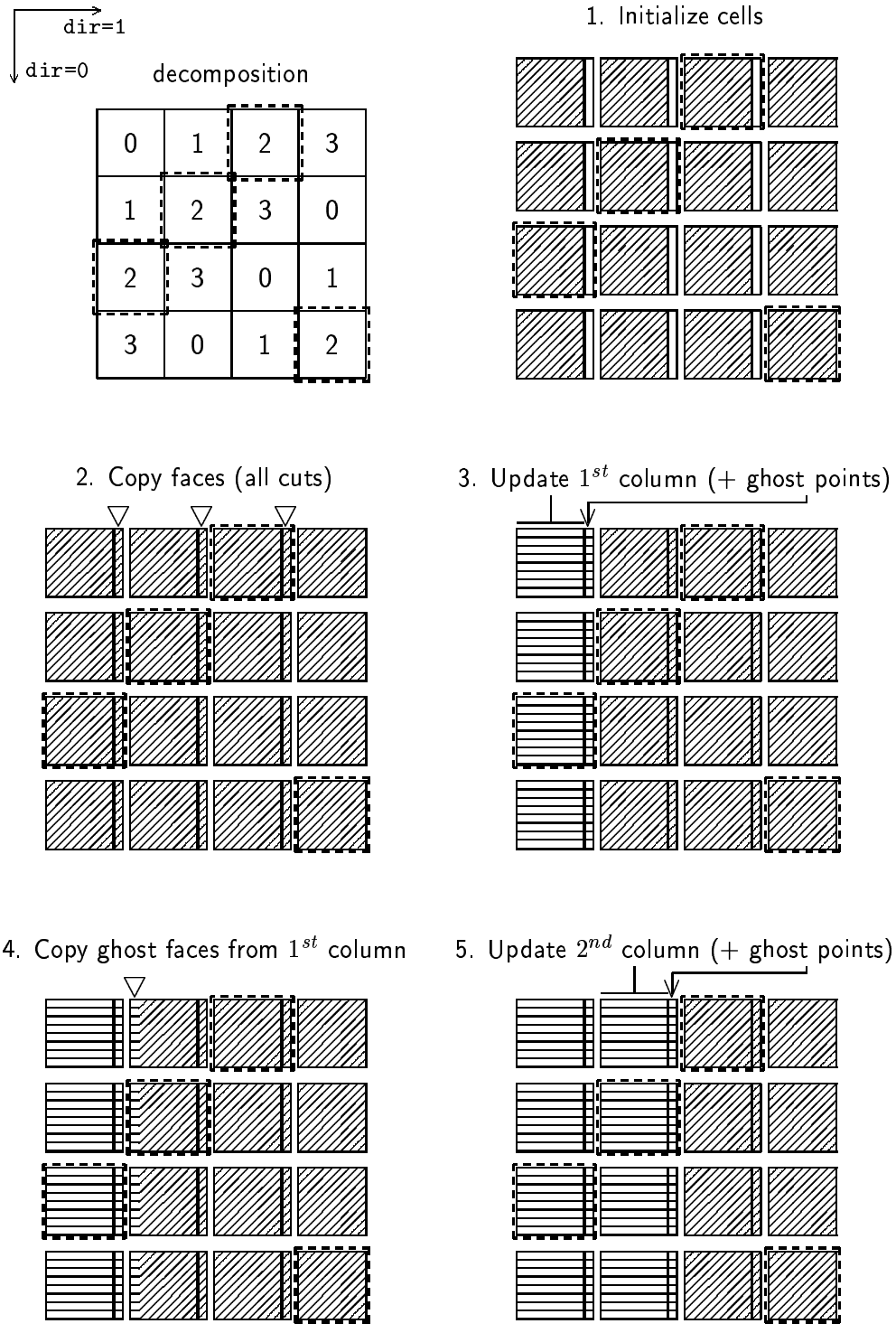


Figure 6.6: First five phases of ADI forward elimination on multi-partitioned grid

to be written as ghost point values enables us to retain the original structure of the loop body, and saves us from having to rely on expensive implicitly invoked communications.

The next optimization is to eliminate all calls to `CHN_Assign`, and replace global with local indexing by subtracting from each global coordinate the starting grid index of the cell (see Section 5.3, page 60). Another optimization would be to rewrite the loop over all the cells in the grid as a loop over only those cells owned by the calling processor. This would require determining first the proper visiting order of owned cells by individual processors.

Communication volume can be reduced by selecting from `lhs` only those elements that need to be copied to neighboring cells, i.e. by defining non-trivial tensor masks. In the pre-loop call to `CHN_Copy_faces` the lower-diagonal and pivot elements (tensor indices 0 and 1) of `lhs` need to be transferred to the "left" neighboring cells, whereas in the call to `CHN_Copy_ghost_faces` within the loop only the pivot element is required by "right" neighbors. We call the corresponding tensor masks `pivot_plus_lower` and `pivot` respectively. They are defined as follows:

```
CHN_Create_tensor_mask(*pivot_plus_lower,lhs_)
CHN_Create_tensor_mask(*pivot,lhs_)
CHN_Set_tensor_mask(pivot_plus_lower,0)
CHN_Set_tensor_mask(pivot_plus_lower,1)
CHN_Set_tensor_mask(pivot,1)
```

Finally, it is possible to eliminate the entire pre-loop communication (`CHN_Copy_faces`) by a simple restructuring of the loop body. This is left as an exercise to the reader.

6.4.2 C-grid flow-through conditions

Scientific programs for realistic problems need to accommodate realistic boundary conditions. Some of these are notoriously difficult to implement using message passing techniques, and virtually impossible to implement efficiently using any other technique. We take as an example the C-grid flow-through conditions for airfoil computations. The physical problem is shown schematically in Figure 6.7a.

A single structured grid is 'wrapped' around the airfoil, such that the computationally distinct grid line segments a-b and d-c, as indicated in Figure 6.7b, coincide in physical space. To ensure single-valuedness of the solution along this cut, numerical analysts usually compute the average of the flow solutions immediately above and below the cut and store this average at both grid points that physically coindice on the cut. Here it is assumed that grid points do exactly match up along the cut, that grid lines pass through it at right angles, and that the grid spacing normal to the cut is symmetrical around the cut.

The problem with this technique is that the two physically close contributors ('p' and 'q') to the value of a point 'x' on the cut are not adjacent in computational space. They may reside on the same processor, or on different processors. Nonetheless, their values must be combined to compute the average. To make matters worse, it is possible that contributions for one processor come from more than one other processors, even in the case of the simple uni-partition decomposition. It is also possible that some contributions are local, while others are remote.

We first give the serial code for a grid of `ni`$\times$`nj` grid points. For simplicity we will only impose the flow-through condition on a scalar variable `rho`. The cut extends from `i=0` to `i=icut` on grid line `j=0`. The reflected side of the cut, therefore, extends from `i=ni-icut-1` to `i=ni-1`.

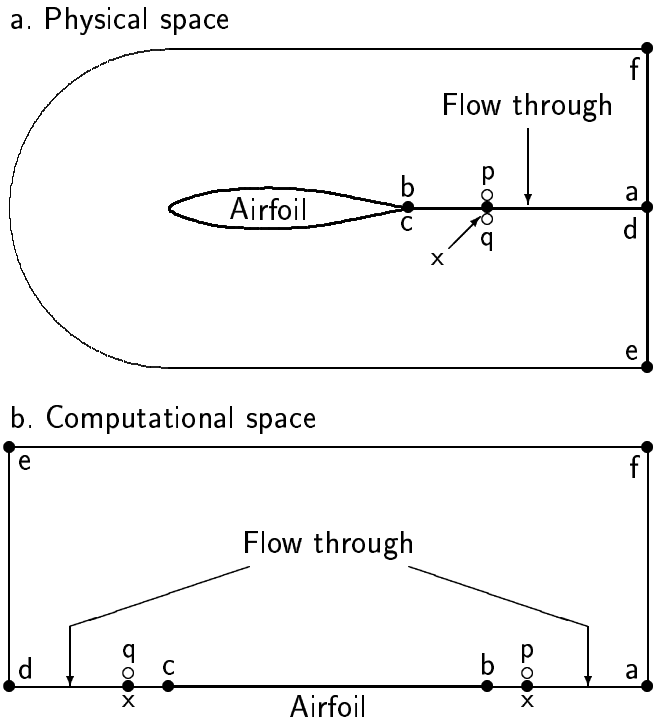


Figure 6.7: C-grid with flow-through condition along branch cut

```
#define rho(i,j)  rho[(j)*ni+i]
float  avg[ni];

for (i=0; i<=icut; i++) {
  avg[i] = 0.5*(rho(i,1)+rho(ni-1-i,1);
  rho(i,0) = rho(ni-1-i,0) = avg[i];
}
```

The first distributed version of this code using Charon is very similar.

```
float  avg[ni];

for (i=0; i<=icut; i++) {
  avg[i] = 0.5*(CHN_Float_value(rho_,i,1)+CHN_Float_value(rho_,ni-1-i,1));
  CHN_Assign(CHN_Address(rho_,i,0),avg[i]);
  CHN_Assign(CHN_Address(rho_,ni-1-i,0),avg[i]);
}
```

Here we have again used the convention of associating an array with a distribution of the same name, suffixed by and underscore character. Notice the use of the auxiliary array avg. It is a global variable, meaning that all processors assign the same values to its elements. This is due to the broadcast nature of CHN_<type>_value. Because there is no distinction between global and distributed values, we can use avg in the CHN_Assign statements to fill elements of a distributed variable.

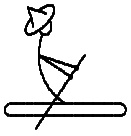
Sometimes the above code is all that is needed, especially if the number of grid points and the number of processors in the communicator are small. But if higher performance is required, for example in 3D computations, where the flow-through condition has to be applied to a whole plane of points, we need to aggregate communications. There are many strategies possible. We start with a simple one, in which one processor (root) claims responsibility for implementing the boundary condition. The processor rank is stored in variable my_rank.

```
float  avg1[ni], avg2[ni];
int start1[2], start2[2], end1[2], end2[2];

start1[0]=0; end1[0]=icut;
start2[0]=ni-1-icut; end2[0]=ni-1;
start1[1]=end1[1]=start2[1]=end2[1]=1;
CHN_Get_tile(rho_, root, avg1, start1, end1, start1, start1, end1);
CHN_Get_tile(rho_, root, avg2, start2, end2, start2, start2, end2);
if (my_rank .eq. root) for (i=0; i<=icut; i++) {
  avg1[i] = 0.5*(avg1[i]+avg2[ni-1-i]);
  avg2[ni-1-i] = avg1[i]
}
start1[1]=end1[1]=start2[1]=end2[1]=0;
CHN_Put_tile(rho_, CHN_ALL, root, avg1, start1, end1, start1, start1, end1);
CHN_Put_tile(rho_, CHN_ALL, root, avg2, start2, end2, start2, start2, end2);
```

Although the arrays avg1 and avg2 have only one subscript, they are interpreted by the communication routines as multi-dimensional arrays. By setting the starting and ending values for the second coordinate equal for both arrays, their effective dimensionality is reduced. Note also that we set insertion and extraction points equal to the starting indices of the local arrays (which are in turn equal to the starting grid indices of the tile). Consequently, copying of distributed data starts at the very beginning of the auxiliary arrays.

The latest implementation scales much better than the first, and will be sufficient for most applications. Further optimizations may be achieved by engaging more processors in the evaluation of the averages. Using Charon query functions, each processor can determine which of the contributing values are local, and can then request remote 'counterparts' to be fetched. This reduces the amount of data communicated by a factor of two, and also spreads the computational load more evenly. However, the logic becomes significantly more complex, and this can only be justified if the amount of work spent on the boundary conditions is substantial in comparison with the work on the interior grid points. Often it is not.

Chapter 7



Input/output

Charon does not at present feature parallel I/O, although the MPI 2 standard would make the implementation of such functionality straightforward. The reason for this omission is the relatively small number of sites where MPI 2 is currently installed. Nonetheless, application programmers will often want to read or write distributed arrays (distributions). One way to accomplish this is to define for each distributed array for which I/O is required a distribution based on so-called solo-partition section and decomposition data structures (see Chapter 3). Reading from disk is performed by using standard I/O on the processor that owns the solo-partition, and calling `CHN_Redistribute` to scatter the contents array to the target distribution. Writing to disk is the reverse of this process.

While this approach is fairly straightforward and relatively efficient compared to many independent read and write accesses by all processors owning part of the distributed array, it does require some extraneous coding by the programmer to define auxiliary data structures. Another strategy is to use `CHN_Read/Write_distribution`. These functions are not any faster than the approach outlined above, because the I/O is still effectively serialized. But they obviate the need to introduce Charon data structures just to access disk files. Another advantage of using these specialized I/O functions is that they give the programmer more control over memory usage. A solo-decomposition distributed array requires a fixed amount of storage, but the Charon I/O functions allow the user to specify a maximum buffer size to be used. This buffer size refers to the amount of user space reserved for gathering data before making a regular system I/O call, not the size of the I/O buffer selected by the operating system kernel. Only the root process (rank zero) in the communicator referenced in the Charon grid definition writes to the file, so it is required to have access to the appropriate file descriptor (unit number in Fortran). Other processors need not have opened that file, or may have even opened another file with the same file descriptor or unit number.

Like redistributions, Charon I/O functions are nonlocal and collective (see Section 1.10). The storage format is binary canonically packed. This means that data for a particular grid is stored without holes; any ghost points or array paddings specified in the distribution are ignored. Moreover, the relative position of tensor components and grid indices in the file is the same as

that of the distribution.

```
int CHN_Read_distribution(int distribution, FILE *stream, int buf_size)
CHN_READ_DISTRIBUTION(distribution, unit, buf_size, ierr)
   integer distribution, unit, buf_size, ierr
```

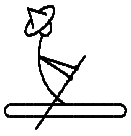
IN	distribution	handle to distribution data structure
INOUT	stream/unit	input file identifier
IN	buf_size	maximum number of data type units to use for buffer space; if CHN_ALL: use as much memory as is available on the system

```
int CHN_Write_distribution(int distribution, FILE *stream, int buf_size)
CHN_WRITE_DISTRIBUTION(distribution, unit, buf_size, ierr)
   integer distribution, unit, buf_size, ierr
```

IN	distribution	handle to distribution data structure
INOUT	stream/unit	output file identifier
IN	buf_size	maximum number of data type units to use for buffer space; if CHN_ALL: use as much memory as is available on the system

If buf_size is set to CHN_ALL, Charon assumes that any memory claimed to read or write the distribution can be allocated on a single processor. Since Charon does not ordinarily release temporary memory because it is likely needed again, using this buffer size value can result in memory problems for certain other operations. If this is suspected, call CHN_Reset_memory (Section 9.5) after performing the I/O.

Chapter 8



Diagnostics

Charon provides a limited number of diagnostics and error modes that may help the programmer debug application programs. Although the query functions listed in the previous chapters give information on the components of all important data structures, it is often convenient to use predefined functions that give a quick overview of entire data structures. These functions print to standard output using a fixed layout. Only the processor whose rank is specified in the parameter list is printed.

```
int CHN_Print_grid_info(int grid, int rank)
CHN_PRINT_GRID_INFO(grid, rank, ierr)
   integer grid, rank, ierr
```

IN	grid	handle to grid data structure
IN	rank	rank (within grid) of processor that prints the information

```
int CHN_Print_section_info(int section, int rank)
CHN_PRINT_SECTION_INFO(section, rank, ierr)
   integer section, rank, ierr
```

IN	section	handle to section data structure
IN	rank	rank (within grid) of processor that prints the information

```
int CHN_Print_decomposition_info(int decomposition, int rank)
CHN_PRINT_DECOMPOSITION_INFO(decomposition, rank, ierr)
   integer decomposition, rank, ierr
```

IN	decomposition	handle to decomposition data structure
IN	rank	rank (within grid) of processor that prints the information

```
int CHN_Print_distribution_info(int distribution, int rank)
CHN_PRINT_DISTRIBUTION_INFO(distribution, rank, ierr)
```

IN	distribution	handle to distribution data structure

IN	rank	rank (within grid) of processor that prints the information

All true Charon procedures return error codes (see the Index). A brief message describing the error can be obtained using function CHN_Print_error, which is modeled after the MPI function MPI_Error_string. This function copies the error message into the variable string, and also returns the number of characters in the message (excluding any string termination character). The user needs to reserve enough space to accommodate the message, whose size is guaranteed not to exceed CHN_MAX_ERROR_STRING.

```
int CHN_Print_error(int err_no, char *string, int *resultlen)
CHN_PRINT_ERROR(err_no, string, resultlen, ierr)
   integer err_no, resultlen, ierr
```

IN	err_no	error code for which information is requested
OUT	string	string containing error information
OUT	resultlen	actual number of characters in message

At any time the programmer can request information on total number of effective assignments done through CHN_Assign (i.e. assignments to actual local addresses), total number of broadcast operations completed to satisfy implicit and explicit remote data requests, and total number of bytes broadcast in the process. The corresponding functions are CHN_Num_assigned, CHN_Num_bcasts, CHN_Num_bytes_bcast, and CHN_Num_bytes_allocated, respectively. In addition, the programmer can obtain the number of MPI send and receive calls issues by Charon functions, and the total number of bytes involved in these respective operations. The corresponding functions are CHN_Num_sends, CHN_Num_recvs, CHN_Num_bytes_sent, and CHN_Num_bytes_recvd. These functions have no error return values.

Another diagnostic is the amount of space in bytes currently allocated as buffer space. As explained in Sections 4.2 and 9.5, Charon manages two types of buffer space (CHN_TEMP_MEMORY and CHN_MVALUE_MEMORY, and both can be queried separately, using CHN_Num_bytes_allocated.

```
int CHN_Num_assigned(void)
integer function CHN_NUM_ASSIGNED()

int CHN_Num_bcasts(void)
integer function CHN_NUM_BCASTS()

int CHN_Num_bytes_bcast(void)
integer function CHN_NUM_BYTES_BCAST()

int CHN_Num_sends(void)
integer function CHN_NUM_SENDS()

int CHN_Num_bytes_sent(void)
integer function CHN_NUM_BYTES_SENT()

int CHN_Num_recvs(void)
integer function CHN_NUM_RECVS()
```

```
int CHN_Num_bytes_recvd(void)
integer function CHN_NUM_BYTES_RECVD()

int CHN_Num_bytes_allocated(int alloc_type)
integer function CHN_NUM_BYTES_ALLOCATED(alloc_type)
   integer alloc_type
```

Error return value: Greatest representable negative integer

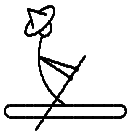
IN	alloc_type	CHN_TEMP_MEMORY or CHN_MVALUE_MEMORY

It is possible that a (pseudo-)collective Charon function will fail on some, but not all, nodes of its communicator. For example, when writing a distributed array to a file, only one node claims space for the output buffer, and this memory allocation may fail, causing the processor to abort the write operation with a non-trivial error code. Intercepting such errors would require broadcasts at every potential asymmetrical point of failure, which would be prohibitively time-consuming. Moreover, since Charon has been constructed with speed in mind, processors that are not actually involved in a certain operation—mostly communications—may skip it, so they would not be able to participate in broadcasts of error conditions. It was therefore decided to allow a failing Charon program to deadlock in some cases when processors that are unaware of a remote error condition try to communicate with the offending processor. If a Charon function is suspected of creating such a situation, the programmer can force the program to abort at any—even relatively benign—error by setting the error mode to CHN_FATAL. The library will attempt to print an error message and signal all processors in MPI_COMM_WORLD to exit gracefully. Setting the error mode to CHN_SERIOUS prints the same error message, but will not cause the program to stop, unless the error is catastrophic. Finally, the programmer may set the error mode to CHN_SILENT (the default) to suppress all error messages.

```
int CHN_Set_error_mode(int mode)
CHN_SET_ERROR_MODE(mode, ierr)
   integer mode, ierr
```

IN	mode	error reporting mode: CHN_SILENT, CHN_SERIOUS, or CHN_FATAL

Chapter 9



Special topics

In this chapter we discuss certain features of the Charon library that may enhance performance, and that make it possible to accommodate a wider range of common coding practices in scientific programs (mostly in Fortran).

9.1 Overindexing

Performance of vector computers can often be improved by increasing the length of vectorizable loops. This provides higher utilization of vector registers, and higher computational throughput. In the case of nested loops, only one loop structure—the innermost—can generally be vectorized. Whereas optimizing compilers can often restructure loop nests to make sure that the inner loop is indeed vectorizable, they can usually not increase its length.

In scientific applications on structured grids, the length of the loop in a loop nest is usually dictated by the size of the grid in the corresponding coordinate direction. Hence, it would appear that the programmer cannot influence the length of vectorizable loops either, unless the grid size is changed as well. Adept Fortran programmers have gotten around this problem by exploiting the weak type checking of most compilers. A Fortran 77 program is incorrect, strictly speaking, if the number and extent of the array dimensions of formal and actual parameters of a subroutine or function differ. But if formal and actual parameters are defined in different files, compilers usually cannot detect index mismatches. Even if they can, most do not issue an error.

A programmer can therefore pass, for example, as an actual parameter an array with three indices to a subroutine expecting a one-dimensional array. Like the Charon distribution data structure, the Fortran dimension statement in the subroutine offers a *structuring interpretation* of the memory space pointed to by the (first element of) the actual parameter. If so desired, the programmer can change the number and extent of the dimensions of the actual parameter inside the subroutine. This technique, when applied to merge several of the leading indices of a multi-dimensional array to create a lower-dimensional array with increased leading dimension, is called overindexing.

In the following example, the subroutines using overindexing are functionally equivalent to that written in the canonical form. But they may obtain significantly higher speed on a vector computer, because of the increased lengths of their inner loops.

```
real a(nx,ny,0:nz+1), b(nx,ny,nz)

call avg_canonical(a,b,nx,ny,nz)
call avg_overindex1(a,b,nx,ny,nz)
call avg_overindex2(a,b,nx,ny,nz)
...

subroutine avg_canonical(a,b,nx,ny,nz)
real a(nx,ny,0:nz+1), b(nx,ny,nz)

do  k = 1, nz
  do  j = 1, ny
    do  i = 1, nx
      b(i,j,k) = 0.5*(a(i,j,k+1)+a(i,j,k-1))
    end do
  end do
end do
return
end

subroutine avg_overindex1(a,b,nx,ny,nz)
real a(nx*ny,0:nz+1), b(nx*ny,nz)

do  k = 1, nz
  do  ij = 1, nx*ny
    b(ij,k) = 0.5*(a(ij,k+1)+a(ij,k-1))
  end do
end do
return
end

subroutine avg_overindex2(a,b,nx,ny,nz)
real a(nx*ny,1,0:nz+1), b(nx*ny,1,nz)

do  k = 1, nz
  do  ij = 1, nx*ny
    b(ij,1,k) = 0.5*(a(ij,1,k+1)+a(ij,1,k-1))
  end do
end do
return
end
```

Routine avg_overindex1 merges the second of the original array subscripts with the first, and drops it in subsequent array references. Routine avg_overindex2 does the same, but effectively

retains the second subscript with a dummy array dimension of size one. Notice that we cannot also merge the third array subscript, because of the shifts applied to it.

Charon supports the form of merging of array subscripts exemplified by `avg_overindex1`, through function `CHN_Set_num_fused_subscripts`. That is, we may specify for a distribution a number of leading array subscripts to be fused, and subsequently drop reference to the fused subscripts from `CHN_Address` and `CHN_(M)Value` parameter lists. Leading subscripts are those with the smallest memory stride (leftmost in Fortran, rightmost in C). Fused subscripts may be grid indices, tensor indices, or a mixture of both.

```
int CHN_Set_num_fused_subscripts(int distribution, int num_fused)
CHN_SET_NUM_FUSED_SUBSCRIPTS(distribution, num_fused, ierr)
   integer distribution, num_fused, ierr
```

INOUT	distribution	handle to distribution data structure
IN	num_fused	number of leading subscripts to be fused

Below we show how the overindexed code on page 90 can be written in Charon. Unlike in the serial code, it is not necessary to use the subroutine interface to redimension the distributed variables (but it is allowed, of course).

```
  call CHN_SET_NUM_FUSED_SUBSCRIPTS(a_,1,ierr)
  call CHN_SET_NUM_FUSED_SUBSCRIPTS(b_,1,ierr)

  do  k = 1, nz
    do  ij = 1, nx*ny
      call CHN_ASSIGN(CHN_ADDRESS(b_,ij,k),0.5*(CHN_REAL_VALUE(a_,ij,k+1)+
 $                                              CHN_REAL_VALUE(a_,ij,k-1)),ierr)
    end do
  end do
  return
  end
```

Careful attention should be given to the value of `num_fused`. It refers to the number of subscripts that will be *dropped* after merging with the leading subscript. Hence, a value of one for `num_fused` means that subscript one will be fused with subscript zero (the leftmost in Fortran). Fusing subscripts does not change the position of any of the data elements related to a distribution, but only influences how positions are computed from lists of subscripts. Adjacency in memory of array elements in a serial code is usually not preserved in a code distributed using Charon (nor in other parallelization packages, for that matter), so a mechanism is needed to infer from a list of subscripts what is the corresponding grid point and tensor component. This mechanism is provided by `CHN_Set_num_fused_subscripts`. Resetting the number of fused subscripts to zero is performed by `CHN_Unset_fused_subscripts`. The query function `CHN_Num_fused_subscripts` returns, for a given distribution, the number of subscripts that have been fused.

```
int CHN_Unset_fused_subscripts(int distribution)
CHN_UNSET_FUSED_SUBSCRIPTS(distribution, ierr)
```

```
   integer distribution, ierr
```

INOUT distribution handle to distribution data structure

```
int CHN_Num_fused_subscripts(int distribution)
integer function CHN_NUM_FUSED_SUBSCRIPTS(distribution)
   integer distribution
```

Error return value: -1

INOUT distribution handle to distribution data structure

Although overindexing is not very commonly used in C, it is also defined for that language. Needless to say, subscript fusing in Charon does not lead to improved performance on vector machines. The overhead incurred by Charon's global access functions (`CHN_Address`, etc.) completely annihilates any possible performance gain. Moreover, array elements that are adjacent in a serial code need not be adjacent (or even on the same processor) in a distibuted code, nor at a constant stride, so vectorizability is virtually destroyed. Instead, subscript fusing is used simply as a convenience to support legacy code practices. In an optimized parallel program the only overindexing used will bypass the Charon interface.

9.2 Subscript reduction

It is often convenient to drop trailing subscripts (rightmost in Fortran, leftmost in C) of an array if they are to be kept constant during a significant portion of the computation. Assignments and evaluation can then use arrays with fewer subscripts, which may increase program clarity and may simplify subscript computations for the compiler.

This type of subscript reduction should not be confused with using a lower-dimensional scratch array (see Section 3.5.3), since each distinct set of dropped ('frozen') trailing subscripts corresponds to a different subset of the space occupied by the original fully-subscripted array. An example will clarify the difference.

```
  real a(nx,ny,nz), b(nx,ny,nz), scratch1(nx,ny), scratch2(nx,ny)

  do  k = 1, nz

c   for every k a different slice of a and b is accessed
    call loop2d(a(1,1,k),b(1,1,k),nx,ny)

c   each value of k uses the same space in the scratch arrays
    do  j = 1, ny
      do  i = 1, nx
        scratch1(i,j) = a(i,j,k)
      end do
    end do
    call loop2d(scratch1,scratch2,nx,ny)
c   must copy changed data from scratch array
    do  j = 1, ny
```

```
      do  i = 1, nx
        b(i,j,k) = scratch2(i,j)
      end do
    end do
  end do
  ...

  subroutine loop2d(a2d,b2d,nx,ny)
  real a2d(nx,ny), b2d(nx,ny)

  do  j = 2, ny-1
    do  i = 2, nx-1
      b2d(i,j) = -4.0*a(i,j)+a(i+1,j)+a(i-1,j)+a(i,j+1)+a(i,j-1)
    end do
  end do
  return
  end
```

The function CHN_Set_fixed_subscripts is used to freeze a number of trailing subscripts at constant values and henceforth drop them from parameter lists in the global access functions (CHN_Address, etc.). The set of frozen subscripts may be spatial indices, tensor indices, or a mixture of both. Of course, we may only specify legal values for these subscripts. We note that trailing subscripts are those with the largest array stride (rightmost in Fortran, leftmost in C).

```
int CHN_Set_fixed_subscripts(int distribution, int num_fixed, ...)
CHN_SET_FIXED_SUBSCRIPTS(distribution, num_fixed, sbs0, sbs1, ..., ierr)
   integer distribution, num_fixed, sbs0, sbs1, ..., ierr
```

INOUT	distribution	handle to distribution data structure
IN	num_fixed	number of subscripts to be frozen
IN	sbs0	value of leading subscript to be frozen
IN	sbs1	value of next-higher-stride subscript to be frozen
IN	...	values of subsequent subscripts to be frozen

Note that the values of the frozen subscripts are listen in order of increasing memory stride (left to right in Fortran, right to left in C). Fixing trailing subscripts and fusing leading subscripts can be applied to the same distribution, as long as there is at least one free subscript left. As an example we show how the first part of the above serial code fragment can be written using Charon functions. Again, no subroutine interface is required to re-index the distributions.

```
  do  k = 1, nz

    call CHN_SET_FIXED_SUBSCRIPTS(a_,1,k,ierr)
    call CHN_SET_FIXED_SUBSCRIPTS(b_,1,k,ierr)

    do  j = 2, ny-1
      do  i = 2, nx-1
        call CHN_ASSIGN(CHN_ADDRESS(b_,i,j),-4.0*CHN_REAL_VALUE(a_,i,j)+
```

```
$                       CHN_REAL_VALUE(a_,i+1,j)+CHN_REAL_VALUE(a_,i-1,j)+
$                       CHN_REAL_VALUE(a_,i,j+1)+CHN_REAL_VALUE(a_,i,j-1),ierr)
      end do
    end do
  end do
```

Resetting the number of frozen subscripts to zero is accomplished by the function `CHN_Unset_fixed_subscripts`. The query functions `CHN_Num_fixed_subscripts` and `CHN_Fixed_subscript` return for a given distribution the number of frozen subscripts, and the actual value of a frozen subscript, respectively. In the latter case the sequence number of the frozen subscript must be specified relative to the first frozen subscript, not the first subscript of the original distribution.

```
int CHN_Unset_fixed_subscripts(int distribution)
CHN_UNSET_FIXED_SUBSCRIPTS(distribution, ierr)
   integer distribution, ierr
```

INOUT	distribution	handle to distribution data structure

```
int CHN_Num_fixed_subscripts(int distribution)
integer function CHN_NUM_FIXED_SUBSCRIPTS(distribution)
   integer distribution
```

Error return value: -1

IN	distribution	handle to distribution data structure

```
int CHN_Fixed_subscript(int distribution, int index)
integer function CHN_FIXED_SUBSCRIPT(distribution, subscript)
   integer distribution, subscript
```

Error return value: greatest representable negative integer

IN	distribution	handle to distribution data structure
IN	subscript	sequence number of subscript within list of frozen subscripts

9.3 Communication caching and optimization

The communication functions `CHN_Copy_faces`, etc., discussed in Chapter 6, are capable of complex data rearrangements and transfers. They act on distributions of arbitrary tensor rank, spatial dimension, number of ghost points, cell ownership pattern. Some special cases are recognized, such as the predefined uni-, multi-, and solo-partitions, and the communications procedures involving them have been optimized to take advantage of their known structure. In general, however, the determination of what data needs to be copied and which processors should send and receive requires a nontrivial amount of logic.

Most scientific computing programs on structured grids consist of large numbers of iterations during which essentially the same operations are applied to the base data set. Hence the overhead of setting up complicated communications could be amortized over the course of the computation if somehow the data structures related to the communications could be stored instead of recomputed each time. In other words, performance improvement might be obtained through

communication caching. This is indeed possible in Charon, through the functions CHN_Begin_remember and CHN_End_remember. They are invoked with a unique, user-supplied integer key, and are placed (logically) before and after the communication operation to be cached. They can therefore be viewed as caching brackets.

```
int CHN_Begin_remember(int key)
CHN_BEGIN_REMEMBER(key, ierr)
   integer key

int CHN_End_remember(int key)
CHN_END_REMEMBER(key, ierr)
   integer key
```

IN	key	unique, user-supplied, non-negative integer

The programmer can 'forget' a caching bracket corresponding to a specific key by invoking CHN_Delete_key. This function releases any memory claimed to save the data structures related to the bracket and makes the key available again for other communication caching. It should not be called inside another communication bracket.

```
int CHN_Delete_key(int key)
CHN_DELETE_KEY(key, ierr)
   integer key, ierr
```

IN	key	unique, user-supplied, non-negative integer

Charon caching brackets work as follows. Upon first entry to the bracket, the key is recognized as not in use, and an entry is created for it in a table of keys. Next, the regular communication operation is carried out, but a record is kept of all atomic operations involved, until the CHN_End_remember function is called (using the same key value). This record is linked to the key. Upon subsequent entries to the bracket, CHN_Begin_remember recognizes the key as previously defined, the corresponding record is retrieved, and its atomic operations are carried out at high speed. The original bracketed communication, with all its tests and logic, becomes a void operation. An implication of this technique is that once the bracket has been initialized, it can be used in other parts of the code, even without including the original communication, but only if all the parameters of the communication are exactly the same. Another implication is that caching brackets cannot be nested.

Charon communication caching is different from that provided by MPI for asynchronous, nonblocking communications (where it is called a persistent communication request) in three respects.

First, no initialization is necessary. Recall that in MPI a persistent communication request is created by a call to MPI_Send_init or MPI_Receive_init, which returns a handle to an MPI_Request data structure. This handle is used in subsequent calls to MPI_Start and MPI_Wait to commence and finalize a communication. As a consequence, the MPI programmer wishing to convert a regular communication into a persistent request needs to isolate the concerned communication from an iterative code segment to create the request exactly once, whereupon it can be used repeatedly. This implies a sometimes cumbersome change in program structure.

Charon caching brackets, however, can be inserted in the program in the location desired, without any additional change. The user is responsible for providing a unique, positive integer to be used as a key. This may conflict with the keys used inside libraries, so it is generally safer for the programmer to obtain an unused key value from Charon through the function `CHN_Unused_key`. Keys obtained through this function are numbered `CHN_MIN_SYSTEM_KEY` and up.

```
int CHN_Unused_key(void)
integer function CHN_UNUSED_KEY()
```
Error return value: `none`

The second difference between Charon and MPI communication caching lies in the execution modes. Creating a persistent communication request is a local, non-collective operation, and a regular communication request on one processor can safely be matched by a persistent communication request on another. Charon caching brackets are local, but collective. They must be used on all processors involved in the communication, although the keys need not be identical on all processors. The reason for this is that Charon may reorder and merge some of the stored atomic operations of the communication in order to improve performance. If this optimization takes place on some processors, but not all, deadlock or—possibly fatal—execution errors may result.

The third respect in which Charon communication caching differs from persistent communication requests is that a single pair of caching brackets may enclose multiple Charon communications. All atomic operations occurring within the communications between the brackets are strung together, and are executed at high speed upon the next entry to the bracket. This saves the programmer from having to define a different key for every communication in a frequently recurring sequence of operations. It also allows Charon to optimize across a larger number of atomic operations, potentially leading to significant performance improvement.

Consider, for example, several consecutive calls to `CHN_Copy_faces_all`, involving different distributions based on the same decomposition. This would ordinarily trigger multiple communications between the same sender and receiver. They may be merged (i.e. messages may be aggregated) if they are enclosed in the same caching bracket. Any sequence of Charon communications can be contained in a single pair of caching brackets, but only atomic operations involving the same elementary data type can be merged.

Other operations, including user-functions, other Charon functions, arithmetic operations, and MPI communications, may be enclosed within caching brackets, but this practice should be avoided. Even without optimization, Charon may reorder statements, because upon the second entry to the bracket, all atomic operations relating to the Charon communications will be executed first, even if in the original logic these communications were interspersed with other operations. An example of communication caching is shown in the the pipeline code on page 105.

If an error occurs in a Charon function inside a caching bracket while it is being defined, the entire bracket is marked as dead, even though some other functions inside the bracket may complete successfully. Subsequent entry into a dead bracket will cause Charon to skip all the functions inside silently.

9.4 Shared solo-partition decomposition

One of the foundations of Charon is that a point in a grid has exactly one owner processor. The functions defined in Section 3.3 make sure that each Cartesian-product subset of the grid is assigned to a unique processor. However, there is one situation in which it is advantageous to have the Cartesian subsets shared among *all* processors. That is when a programmer wishes to switch between serial and parallel processing within a single program in the course of converting a serial code (see Section 10.1.2). The Charon function that creates a shared decomposition out of a section data structure is `CHN_set_sharedpartition_owners`.

```
int CHN_Set_sharedpartition_owners(int decomposition)
CHN_SET_SHAREDPARTITION_OWNERS(decomposition, ierr)
   integer decomposition, ierr
```

`INOUT decomposition` handle to decomposition data structure

There is only one type of Charon section that is eligible for sharing, and that is the solo-partition section (single cell). The only allowable operation on distributions defined on a shared-partition decomposition is redistribution. The result of mapping any regular distribution to a shared-partition distribution is the same as that of mapping to a distribution based on a solo-partition decomposition (single owner), except that now *all* processors receive the same data; it is broadcast to all processors in the communicator. The advantage is that all processors can henceforth execute serial code, so that it is not necessary to exclude any processors, as would have to be the case if a solo-partition decomposition were used.

The result of mapping a shared-partition distribution to a regular distribution is identical to that of mapping a solo-partition distribution owned by processor zero to the regular distribution. Since serial code does not make use of the processor number, all processors perform the same operations on data contained in the shared-partition distribution. Therefore it does not make a difference which processor furnishes the data for redistribution to a regular distribution.

9.5 Temporary memory allocated

Charon communication functions use temporary memory that is allocated by a fast special-purpose scheme. The allocation strategy is based on the premise that many applications for which Charon is useful are iterative, and that the same communication functions are called many times. Rather than allocating and deallocating all the time, Charon keeps track of the exact amount of temporary memory needed at any one time in the program. If that amount is less than the temporary memory allocated earlier (and no longer in use), then that buffer will be reused. If the amount is more than was allocated before, the previous buffer is released and a new one is allocated. Consequently, while the temporary memory allocated never grows to a size larger than is required by the most demanding Charon communication function, it is never permanently released by the library. If storage is at a premium the user can release the buffer space explicitly by using `CHN_Release_memory` and specifying the type of memory involved: `CHN_TEMP_MEMORY`. This has no influence on the correctness of the code, and is not very expensive.

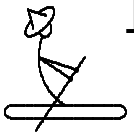
Another type of system-allocated memory relates to `CHN_Mvalue`. As explained in Section 4.2, this type of memory is not volatile and does not get reused. It has to be released explicitly if the

contents is no longer needed and the programmer wishes to free the space. This is accomplished by calling CHN_Release_memory and specifying the type of memory involved: CHN_MVALUE_MEMORY. The result is that *all* buffers claimed by function CHN_Mvalue are released.

```
int CHN_Release_memory(int alloc_type)
CHN_RELEASE_MEMORY(alloc_type, ierr)
   integer alloc_type, ierr
```

IN	alloc_type	buffer type: CHN_TEMP_MEMORY or CHN_MVALUE_MEMORY

Chapter 10



Programming examples

10.1 General programming tips

Charon offers a powerful set of functions for manipulation of distributed variables. Some programming techniques emerge from experience that make these functions easier to use, and that expedite parallel program development and conversion.

10.1.1 Aliases

Many Charon functions have rather long names, and sometimes carry more arguments than is desirable for program clarity. Aliases may be defined that shorten names and parameter lists. These are especially useful for the most occurring Charon functions,namely those of `CHN_Assign`, `CHN_Address`, and `CHN_<type>_value`. Since they will be removed from the final program text anyway, performance of the tuned program is not affected by introduction of a set of wrapper routines. We present two types of handy wrapper routines in Fortran. They both assume that all user definitions of distributed variables have been placed in common blocks in a header file named `my_pars.incl`. In C macros may be utilized that do not add any overhead.

1. For each distributed variable `vl_` occurring on the left hand side of an assignment statement, introduce a subroutine `assign_vl`. Assuming the single precision variable has three subscripts, define the subroutine as follows.

```
subroutine assign_vl(i,j,k,val)
include 'my_pars.incl'
integer i, j, k, ignore_error
real    val

call CHN_Assign(CHN_Address(vl_,i,j,k),val,ignore_error)
return
end
```

2. For each distributed variable vr_ whose value is needed in an expression, introduce a function vr_val. Assuming that the single precision variable also has three subscripts, define the function as follows.

```
real function vr_val(i,j,k)
include 'my_pars.incl'
integer i, j, k

vr_val = CHN_REAL_VALUE(vr_,i,j,k)
return
end
```

Using such definitions, the first version of the Charon code implementing the 3D seven-point star stencil computation on page 58 can be written in a significantly terser form, which is also closer to the original serial version. Evidently, savings would be greater when dealing with double precision variables.

```
do  k = 2, nsize(2)-1
  do  j = 2, nsize(1)-1
    do  i = 2, nsize(0)-1
      call assign_r(i,j,k, -6.0*a_val(i,j,k)  + a_val(i+1,j,k)+
$                              a_val(i-1,j,k)+ a_val(i,j+1,k)+
$                              a_val(i,j-1,k)+ a_val(i,j,k+1)+
$                              a_val(i,j,k-1))
    end do
  end do
end do
```

10.1.2 Serial-parallel adaptors

Often it is convenient to leave the bulk of a legacy code in serial mode while concentrating on distributing and parallelizing only one or a few modules at a time. This can be accomplished, as was detailed in Section 6.1.2, using the redistribution facility within Charon. For each distributed variable two distributions are defined. One features the target multi-owner decomposition (e.g. dmo_), the other a single-owner solo-partition decomposition (e.g. dso_) whose layout corresponds to that of the legacy code. A convenient pair of adaptor functions can now be defined as follows, assuming, as before, that all user definitions of distributed variables have been placed in common blocks in a header file named my_pars.incl.

```
subroutine begin_distributed
include 'my_pars.incl'
integer ignore_error

call CHN_REDISTRIBUTE(dmo0_,dso0_,ignore_error)
call CHN_REDISTRIBUTE(dmo1_,dso1_,ignore_error)
call CHN_REDISTRIBUTE(dmo2_,dso2_,ignore_error)
call CHN_REDISTRIBUTE(dmo3_,dso3_,ignore_error)
```

```
call CHN_REDISTRIBUTE(dmo4_,dso4_,ignore_error)
...
return
end

subroutine end_distributed
include 'my_pars.incl'
integer ignore_error

call CHN_REDISTRIBUTE(dso0_,dmo0_,ignore_error)
call CHN_REDISTRIBUTE(dso1_,dmo1_,ignore_error)
call CHN_REDISTRIBUTE(dso2_,dmo2_,ignore_error)
call CHN_REDISTRIBUTE(dso3_,dmo3_,ignore_error)
call CHN_REDISTRIBUTE(dso4_,dmo4_,ignore_error)
...
return
end
```

If a certain code fragment in the middle of the program needs to be parallelized, it is simply bracketed by calls to `begin_distributed` and `end_distributed`. The only caveat is that, upon return to serial mode, only one processor—the owner of the only cell in the solo-partition—owns useful data created in the parallel section. Hence, other processors should not execute any serial code that may create arithmetic exceptions or other failures due to inconsistent data. This situation can be avoided by making sure that all processors receive the same data upon entering the serial mode. This is accomplished by using function `CHN_Set_sharedpartition_owners` to define a special, restricted decomposition whose cells are owned by all processors in the communicator (see Section 9.4).

10.2 Multiple topologically independent grids

Charon helps the user manage the distribution aspects of structured grids within communication domains (MPI communicators). Sometimes it is desirable to define several topologically independent grids, each with their own communication domain. Charon does not support communications between such domains directly; implicitly invoked as well as explicit communications are restricted to the communicators specified in the `CHN_Create_grid` call.

This is usually the behavior the user wants, because it allows modular program development, and limits interference between processors working on different grids. Data exchange between processors in these domains is possible, though, using so-called MPI intercommunicators.

Assume that there are two domains, dom0 and dom1, and information pertaining to a set of points in a grid in dom0 needs to be used in computations on a grid in dom1, for example to provide interpolated values. The data can be accumulated on one or more processors in dom1 through `CHN_Get_tile`, sent to one or more processors in dom2 through a regular MPI communication using the appropriate intercommunicator, and placed in a distributed variable in dom2 through `CHN_Put_tile`. If the data is very irregularly distrbuted across the grid, tiles may not be the best mechanism, and individual data items would have to be placed in and copied from buffers, possibly using Charon query functions to optimize locality.

10.3 Pipelined algorithms

Pipelining is a generally applicable technique for extracting parallelism from scientific problems with difficult data dependencies. It is a truly non-data-parallel concept, and therefore cannot be expressed in systems that only allow data parallel program specifications. We take as an example application the Gauss-Seidel relaxation technique for solving the heat equation on a uniform grid of `ni`×`nj` points.

The method can be summarized as follows. For each point in the grid, replace the value of the temperature `T` by the average temperature of its four closest neighbors, always using the latest updated values available. Temparatures on the boundaries are kept fixed. The result of this relaxation depends on the order in which points are visited. It can be made unambiguous by requiring that no point gets updated before its neighbors whose coordinates are smaller than or equal to the target point's coordinates are updated. The simplest scheme satisfying this requirement visits points in the canonical lexicographical order, as exemplified by the following code.

```
  do  j=2, nj-1
    do  i=2, ni-1
      T(i,j) = 0.25*(T(i+1,j)+T(i,j+1)+T(i-1,j)+T(i,j-1))
    end do
  end do
```

Data parallel implementations of this algorithm have difficulty extracting parallelism, because there is a strong data dependence. The wavefront method proposed by Lamport [17] rearranges the computions such that all points on a diagonal line (a wavefront) defined by `i+j=constant` can be updated independently (and hence simultaneously). We will assume that $\texttt{ni} \leq \texttt{nj}$. Here is the serial program.

```
c loop over the number of diagonals (wavefronts) in the grid
  do  dg=4, ni+nj-2
c   must determine extremal points of the diagonal
    if (dg .le. ni+1) then
      istart = 2
      iend   = dg
    elseif (dg. le. nj+1) then
      istart = dg-ni+1
      iend   = dg
    else
      istart = dg-ni+1
      iend   = ni-1
    endif
    do i = istart, iend
      j = dg-i
      T(i,j) = 0.25*(T(i+1,j)+T(i,j+1)+T(i-1,j)+T(i,j-1))
    end do
  end do
```

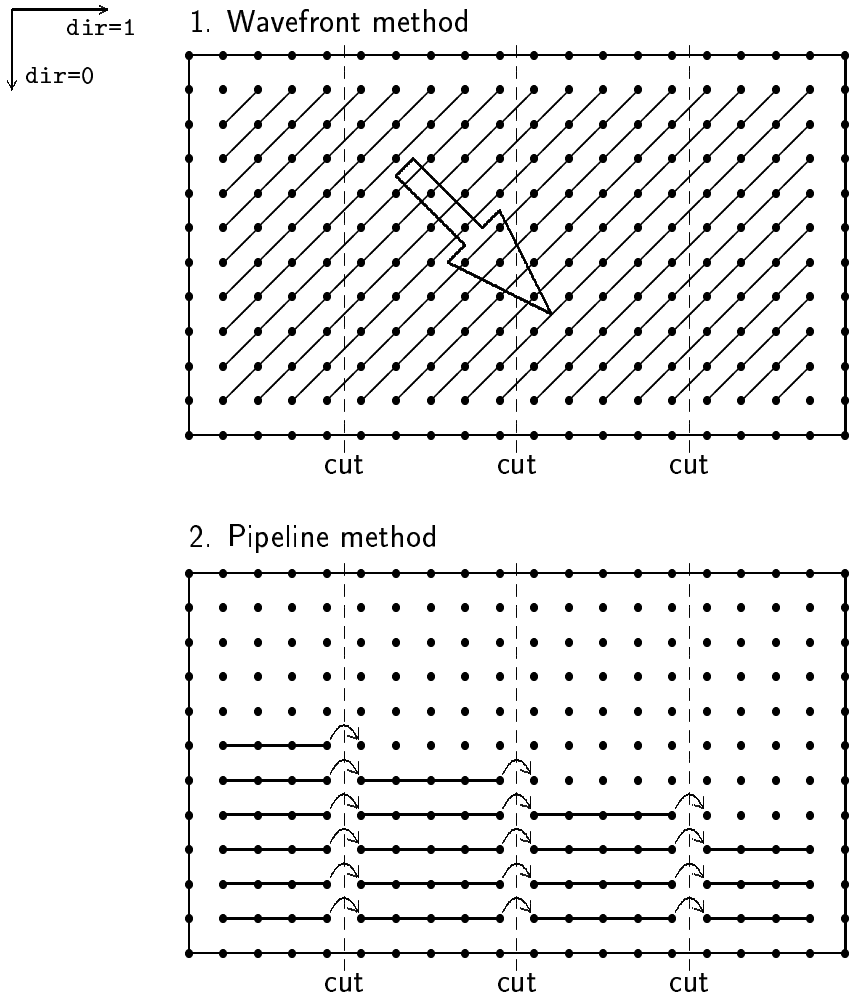


Figure 10.1: Wavefront and pipeline techniques for implementing Gauss-Seidel relaxation method

A graphical representation of the progressing wavefront is shown in Figure 10.1.1. Diagonal lines are the wavefronts, which are updated in the order indicated by the direction of the open arrow. Whereas this technique manages to create an inner loop whose iterations are independent of each other (important for vector machines), it suffers from several defects when applied as a vehicle for parallelization.

First, the length of the inner loop varies during the sweep across the grid, and with it also the amount of extractable parallelism. This induces a load imbalance. Second, the maximum number of processors that can be active at any one time equals `ni-2` (the maximum number of points on a wavefront). The theoretically maximum efficiency of the method is further compromised if a realistic domain decomposition is chosen that minimizes the ratio of interior to boundary points for each cell. For example, if the grid is divided into strips, as indicated in Figure 10.1.1, processors owning cells near the `i`-axis will run out of computational work long before the last processor has finished. This effect would be exacerbated by a decomposition into squares instead of into strips.

A better solution is to pipeline the process. Refer again to the stripwise decomposition of the domain (Figure 10.1.2). The first processor updates all points it owns on the first grid line.

Once the last point on the line segment is updated, it transmits that value (curved arrow) to the processor owning the adjacent strip, which can then start computations. Meanwhile, the first processor starts work on the next grid line. After a while, all processors will be active, each working on different grid lines; the pipeline is filled. Figure 10.1.2 shows the status of the full pipeline after six successive line segments have been updated by the first processor.

The difficulty in devising automatic parallelizing compilers that efficiently implement pipelines lies is the choice of the proper granularity. The above process works properly and exhibts a fully balanced load once the pipeline is full. But if there are not many points on each grid line segment, very frequent communications are needed. In that case if may be better to group a number of line segments together and postpone communications until all segments in a group have been updated. The number of line segments in a group is called the pipeline group factor. While the programmer often has a good grasp of what is a reasonable group factor, compilers have a hard time guessing it [13]. Charon helps the user to program pipelines explicitly, with relatively little effort. We will assume that the group factor `igrp`, set by the programmer, evenly divides the number of interior grid points in the first coordinate direction. Otherwise, some simple preconditioning would be necessary.

Here is the first distributed version of the code for the pipelined algorithm. We assume that the decomposition on which distribution `T_` is based is called `dcmp`. Even though no communications are required, we structure the code such that the points are accessed similar to the way they will be in the parallel code. But, of course, we cannot indicate concurrency in a serial code.

```
c loop over all strips
  do c = 0, CHN_total_num_cells(dcmp)-1
    do grp = 1, ni/igrp
      istart = 2+(grp-1)*igrp
      iend   = istart + igrp-1
      do j=max(2,CHN_CELL_START_INDEX(dcmp,1,c)),
 $         min(nj-1,CHN_CELL_END_INDEX(dcmp,1,c))
        do i = istart, iend
          call CHN_ASSIGN(CHN_ADDRESS(T_,i,j), 0.25*(CHN_REAL_VALUE(T_,i+1,j)+
 $                        CHN_REAL_VALUE(T_,i,j+1)+ CHN_REAL_VALUE(T_,i-1,j)+
 $                        CHN_REAL_VALUE(T_,i,j-1)), ierr)
        end do
      end do
    end do
  end do
```

Now make the communications explicit, so that broadcasts can be suppressed. At the same time, we let only the processor that owns a grid point call the corresponding `CHN_Assign` routine. The assignments are parallelized, but the communications and index calculations are not. Even though in Fortran we could, in principle, still declare `istart` and `iend` as scalars, we choose to dimension them as integer arrays of length 1. This highlights the fact that communication panels are generally vectors, whose length is one less than the number of dimensions of the grid.

```
  integer start(1), end(1)
```

```
c make sure all ghost point values are initialized (old values)
  istart(1) = CHN_ALL
  iend(1)   = CHN_ALL
  call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_LEFT,CHN_ALL,istart,
 $                    iend,ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,my_rank,ierr)

  call CHN_BEGIN_LOCAL(MPI_COMM_WORLD,ierr)
  do c = 0, CHN_total_num_cells(dcmp)-1
    do grp = 1, (ni-2)/igrp
      istart(1) = 2+(grp-1)*igrp
      iend(1)   = istart(1) + igrp-1
      if (CHN_OWNER(dcmp,c) .eq. my_rank) then
        call CHN_BEGIN_GHOST_ACCESS(dcmp,c,ierr)
        do j=max(2,CHN_CELL_START_INDEX(dcmp,1,c)),
 $           min(nj-1,CHN_CELL_END_INDEX(dcmp,1,c))
          do i = istart(1), iend(1)
            call CHN_ASSIGN(CHN_ADDRESS(T_,i,j), 0.25*(CHN_REAL_VALUE(T_,i+1,j)+
 $                          CHN_REAL_VALUE(T_,i,j+1)+ CHN_REAL_VALUE(T_,i-1,j)+
 $                          CHN_REAL_VALUE(T_,i,j-1)), ierr)
          end do
        end do
        call CHN_END_GHOST_ACCESS(dcmp,c,ierr)
      end if
c     send latest data to next strip
      call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_RIGHT,c,istart,
 $                        iend,ierr)
    end do
  end do
  call CHN_END_LOCAL(comm,ierr)
```

The structured communications routines are called by all processors, to make sure that those actively involved in sending and receiving data will be executing them. However, we know that only one pair of processors actually needs to call each routine. The following variation reduces the number of idle communication calls, skips the ownership tests, and simplifies the loop structure. We also drop the calls to the global access functions and revert to local indexing (must use offset gp to skip over ghost points). Communication caching is used to enable further optimzation (see Section 9.3), assuming that the code fragment is executed many times in an iterative solver.

```
  integer start(1), end(1), gp
  parameter (gp=1)

c make sure all ghost point values are initialized (old values)
  istart(1) = CHN_ALL
  iend(1)   = CHN_ALL
  call CHN_BEGIN_REMEMBER(123,ierr)
  call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_LEFT,CHN_ALL,istart,
 $                    iend,ierr)
```

```
  call CHN_END_REMEMBER(123,ierr)

  c = CHN_own_to_global_cell_index(dcmp,0)
  do grp = 1, (ni-2)/igrp
    istart(1) = 2+(grp-1)*igrp
    iend(1)   = istart(1) + igrp-1
c   make sure we receive latest data from previous strip; use unique caching key
    call CHN_BEGIN_REMEMBER(123+2*grp,ierr)
    call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_RIGHT,c-1,istart,
 $                      iend,ierr)
    call CHN_END_REMEMBER(123+2*grp,ierr)
    do jj=max(2,CHN_CELL_START_INDEX(dcmp,1,c)),
 $        min(nj-1,CHN_CELL_END_INDEX(dcmp,1,c))
      j=jj-CHN_CELL_START_INDEX(dcmp,1,c)+gp
      do i = istart(1)+gp, iend(1)+gp
        T(i,j) = 0.25*(T(i+1,j)+T(i,j+1)+T(i-1,j)+T(i,j-1))
      end do
    end do
c   send latest data to next strip; use unique caching key
    call CHN_BEGIN_REMEMBER(123+2*grp+1,ierr)
    call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_RIGHT,c,istart,
 $                      iend,ierr)
    call CHN_END_REMEMBER(123+2*grp+1,ierr)
  end do
```

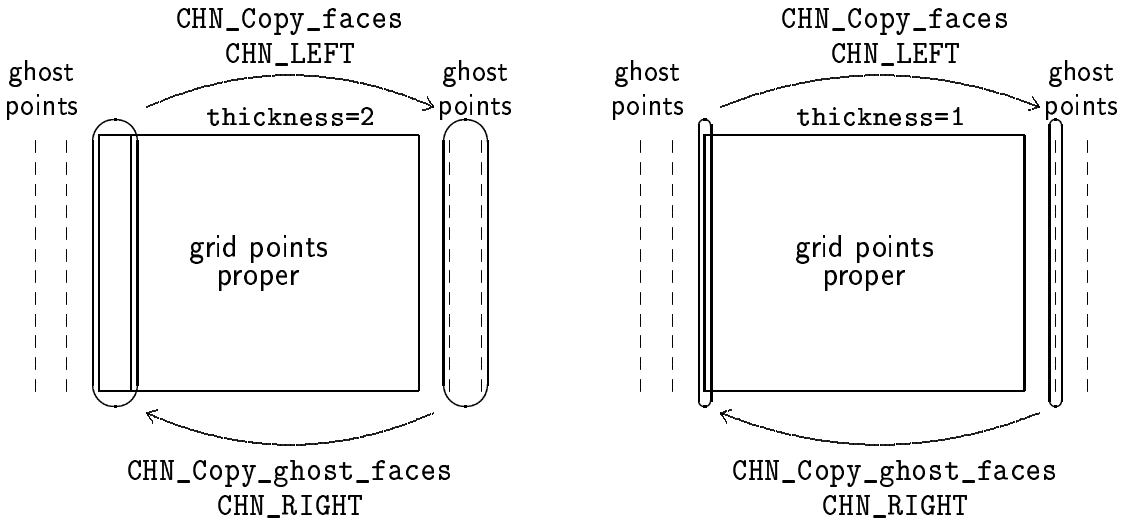
10.4 Periodic solvers

In serial stencil-based programs periodic solvers are usually implemented by employing layers of buffer points around the grid on which the solution is sought. Data from one side of the grid is then copied to the opposite side, to reflect the periodicity. In parallel programs constructed using Charon, the equivalent of buffer points is available through the ghost points. However, ghost point values are not ordinarily available through the global access functions `CHN_Address`, etc., until the programmer explicitly requests them using `CHN_Begin_ghost_access`. This is undesirable during the initial phases of the parallel program development, because it involves more coding and also requires making the domain decomposition visible.

The solution is for the user to define the initial Charon grid large enough to include any points that are to serve as buffer points for the periodic solver. For example, in a periodic, 2D, five-point-star stencil computation the grid would be padded with one point on each side (for a total of two) in the direction of the periodicity(ies). This works fine, except that the speedy structured communication functions that copy values between faces of cells (`CHN_Copy_(ghost)_faces`) cannot be used directly. They move data into and out of ghost points managed by Charon, not those managed by the programmer. So an additional copy of values from Charon-managed to user-managed ghost points would have to follow, which reduces program efficiency.

A shortcut is obtained by using the argument `CHN_PERIODIC_TRUNCATED` instead of `CHN_PERIODIC` in the copy operation. For a copy operation defined with thickness `d` this has exactly

a. CHN_PERIODIC



b. CHN_PERIODIC_TRUNCATED

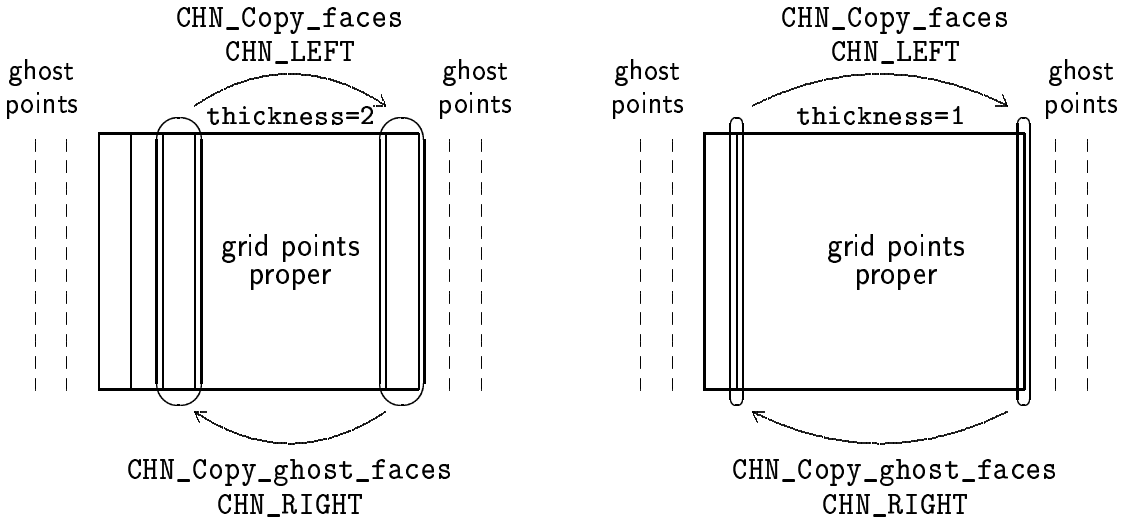


Figure 10.2: Copying periodic cell face values with (bottom) and without (top) truncation for a distribution with two ghost points

the same effect as if the outermost d points of the grid in the coordinate direction of the copy operation (both sides) had been Charon ghost points. Figure 10.2 shows the difference between plain (Tableau a.) and truncated (Tableau b.) periodic data movements for copy operations in the second coordinate direction (cf. Figure 6.2). Because the copying of interior cell faces is not affected by the setting of the periodicity parameter, we merely show the copy result for a solo-partition decomposition. The example distribution has been defined with two ghost points (none are required by the truncated copying).

Truncation only affects the apparent size of the grid in the copy direction. If a problem is periodic in multiple directions, the user may want to restrict the copy panel (see Section 6.1.1) to

exclude those points that fall outside the truncated grid in directions normal to the copy direction.

10.4.1 Explicit periodic boundary conditions

Point-relaxation schemes, such as the point-Gauss-Seidel or point-Jacobi iterative methods, are primarily used for their computational simplicity and efficiency. Save for data dependencies that sometimes complicate parallel implementations, they are as straightforward as explicit methods. This simplicity would be ruined by insisting on an implicit implementation of periodic boundary conditions. More often, periodic boundary conditions are implemented explicitly; boundary values are updated independently on one side of the domain, and copied to the other side.

Consider again the point-Gauss-Seidel problem of Section 10.3. We modify it to be periodic in the j-direction. The grid size is again ni×nj points, but now the first and last points in the j-direction are assigned to user-managed buffer duty. We will assume that the periodic boundary corresponds to the grid lines j=1 and j=nj−1[1].

The algorithm proceeds as indicated schematically in Figure 10.3. Before updating interior grid points, copy data from the right periodic boundary to the left (phase 1), using CHN_PERIODIC_TRUNCATED. The left boundary now consists of user-managed buffer points, while the right does not, so CHN_Copy_faces is the appropriate communication function. Subsequently, update interior points in exactly the same way as in the non-periodic case—including pipelining and copying of data across interior cell faces—but exclude the points on the right periodic boundary (phase 2).

After all interior points have been updated, copy data from the line immediately adjacent to the left periodic boundary (line j=2) to the corresponding points on the opposite side of the grid (phase 3). The source points are now truly interior, and the destination consists of user-managed buffer points, so the appropriate communication function is CHN_Copy_ghost_faces. Finally, update the points on the right periodic boundary, using the data that has just been copied (phase 4).

Below we present the Charon code that implements this solver for the stripwise decomposition of Section 10.3.

```
  integer start(1), end(1), gp
  parameter (gp=1)

c make sure all ghost point values are initialized (old values)
  istart(1) = CHN_ALL
  iend(1)   = CHN_ALL
  call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_LEFT,CHN_ALL,istart,
 $                    iend,ierr)

c copy values to the left periodic boundary
  call CHN_COPY_FACES(T_,CHN_PERIODIC_TRUNCATED,1,CHN_ALL,1,CHN_RIGHT,-1,istart,
 $                    iend,ierr)
```

[1] It is equally well possible to let the periodic boundary correspond to lines j=2 and j=nj. The reader is invited to verify that this can also be parallelized easily, using essentially the same Charon functions.

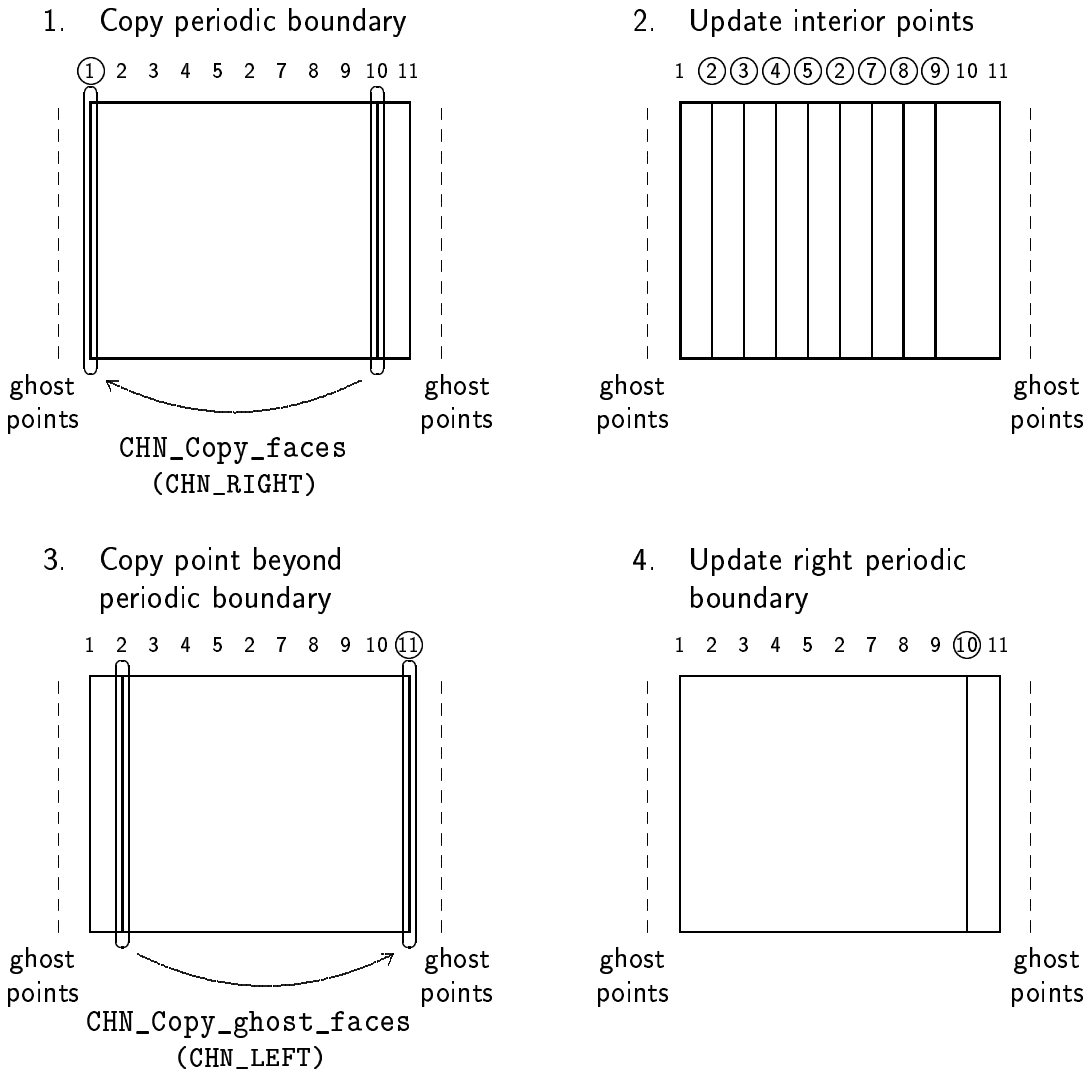


Figure 10.3: Using TRUNCATED copying to implement periodic point-Gauss-Seidel relaxation method

```
c update interior points
  c = CHN_own_to_global_cell_index(dcmp,0)
  do grp = 1, (ni-2)/igrp
    istart(1) = 2+(grp-1)*igrp
    iend(1)   = istart(1) + igrp-1
c   make sure we receive latest data from previous strip
    call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_RIGHT,c-1,istart,
 $                      iend,ierr)
    do jj=max(2,CHN_CELL_START_INDEX(dcmp,1,c)),
 $        min(nj-2,CHN_CELL_END_INDEX(dcmp,1,c))
      j=jj-CHN_CELL_START_INDEX(dcmp,1,c)+gp
      do i = istart(1)+gp, iend(1)+gp
        T(i,j) = 0.25*(T(i+1,j)+T(i,j+1)+T(i-1,j)+T(i,j-1))
      end do
```

```
    end do
c   send latest data to next strip
    call CHN_COPY_FACES(T_,CHN_NONPERIODIC,1,CHN_ALL,1,CHN_RIGHT,c,istart,
 $                      iend,ierr)
  end do

c copy values from the points adjacent to the left periodic boundary
  call CHN_COPY_GHOST_FACES(T_,CHN_PERIODIC_TRUNCATED,1,CHN_ALL,1,CHN_LEFT,-1,
 $                          istart,iend,ierr)

c update the values on the right periodic boundary
  if (c .eq. CHN_Total_num_cells(dcmp)-1) then
    jj = nj-1
    j=jj-CHN_CELL_START_INDEX(dcmp,1,c)+gp
    do i = 1+gp, ni+gp
      T(i,j) = 0.25*(T(i+1,j)+T(i,j+1)+T(i-1,j)+T(i,j-1))
    end do
  end if
```

We make the following observations.

- Copying across a periodic boundary on the right side of the grid in the right direction (`CHN_RIGHT`) moves the data to the opposite side of the grid, i.e. the *left* side. Hence, we use `CHN_RIGHT` to fill the points on the left periodic boundary.

- Similarly, we use `CHN_LEFT` to fill the points adjacent to the right periodic boundary.

- The n cuts of a section are numbered 0 through $n-1$. The virtual periodic cuts on both sides of the grid have sequence numbers -1 and n. Since they logically coincide, specifying either -1 or n for the cut number results in the same copy operation.

10.4.2 Implicit periodic boundary conditions

If the interior point algorithm of the numerical scheme is implicit, it may be desirable to implement periodic boundary conditions implicitly as well. For example, the numerical analyst may use the ADI method of Section 6.4.1 for solving a numerical problem, which results in the solution of (implicit) banded systems for each whole grid line. A periodic boundary condition can be incorporated into the matrix system, but destroys the banded structure of the system by modifying the first and the last few matrix rows.

A common solution is to decompose the modified matrix into submatrices, which can be inverted by solving *twice* a banded system very close to the original. This banded system can be solved using the method described in Section 6.4.1. In the process, some data has to be transferred between the opposite sides of the grid, which can again be accomplished by the Charon cell face copy routines with `CHN_PERIODIC_TRUNCATED`.

10.5 Multigrid

One of the most efficient numerical methods for solving discretized partial differential equations is multigrid. It is based on the notion that many classical iterative schemes are effective at removing from the solution error components whose frequency is high relative to the grid spacing, but are poor at damping low-frequency components. What is a low-frequency wave on a fine grid is a high-frequency wave on a coarse grid, so it can be beneficial to employ a coarse grid to remove the errors of the largest wave length, and interpolate the resulting solution to a finer grid to remove errors of smaller wave length. A hierarchy of grids can be constructed, each finer than the previous, so that errors of all wave lenghts are ultimately removed in the most efficient way.

Many iterative schemes can be used for error reduction on individual grids, and many refinement strategies are possible. The most common approach is to divide all grid spacings in two when moving to the next finer level. We will assume that a relaxation scheme for an individual level has been chosen, and that it has been parallelized using Charon functions. The difficulty is now to relate solutions and residuals on different grid levels to each other, while also respecting locality as much as possible. By that we mean that all grid points coincident in space have the same owner processor, regardless of the level of refinement. This is an important property, because it allows us to interpolate to higher and restrict to lower levels of refinement without concern about communications.

A possible solution is to define grids at all levels to have the same number of points as the very finest, and skip any points not used. This is convenient, but very wasteful of space. Moreover, on a cache-based computer the relative sparsity of data on the coarser grids would cause a substantial loss in performance. A better way is to use Charon's fine-tuning capabilities to make sure that locality is obtained.

We will examine a 3D example. The grid has $(n_x(2^k+1)) \times (n_y(2^k+1)) \times (n_z(2^k+1))$ points, where k indicates the level of refinement. The smallest grid on which a solution is sought has size $n_x \times n_y \times n_z$, which corresponds with multigrid level $k = 0$. Assume that all grid coordinates start at zero. We first consider the case where this smallest grid can be partitioned uniformly—all processors receive (almost) the same number of points—using the Charon functions of Section 3.2.

We refer to the value of cut i in coordinate direction d at grid level k as $c^k_{d,i}$. Hence, $c^0_{2,1} = n$ means that on the coarsest grid the second cut in the z-direction (third coordinate direction) is placed between points $n-1$ and n. Now assume a section has been completed for the coarsest grid. The requirement that coincident points in space at different levels of refinement be assigned to the same processor implies the following consistency condition for cuts: $c^k_{d,i} = 2c^{k-1}_{d,i} = 2^k c^0_{d,i}$. Thus, all cuts at higher levels of refinement can be related to cuts at the lowest level of refinement. Regardless of how the coarsest grid has been partitioned, the following code fragment will create decompositions that observe the above consistency rule.

```
integer grid(0:KMAX), sect(0:KMAX), dcmp(0:KMAX), k, dir, cut, cell
do k = 1, KMAX
  call CHN_CREATE_GRID(grid(k),CHN_GRID_COMM(grid(0)),3,ierr)
  do dir = 0, 2
    call CHN_SET_GRID_SIZE(grid(k),dir,2*CHN_GRID_SIZE(grid(k-1),dir)-1,ierr)
  end do
  call CHN_CREATE_SECTION(sect(k),grid(k),ierr)
```

```
    do dir = 0, 2
      call CHN_SET_NUM_CUTS(sect(k),CHN_NUM_CUTS(sect(0),dir),ierr)
      do cut = 0, CHN_NUM_CUTS(sect(0),dir)-1
        call CHN_SET_CUT(sect(k),dir,cut,2*CHN_CUT(sect(k-1),dir,cut),ierr)
      end do
    end do
    call CHN_CREATE_DECOMPOSITION(dcmp(k),sect(k),ierr)
    do cell = 0, CHN_TOTAL_NUM_CELLS(dcmp(0))
      call CHN_SET_CELL_OWNER(dcmp(k),CHN_CELL_OWNER(dcmp(0),cell),cell,ierr)
    end do
  end do
```

If the owners of the cells on the coarsest grid have been set using any of the predefined decomposition functions (`CHN_Set_uni/multi/solopartition_owners`), then it is preferable to use the same function for higher levels of refinement, because Charon communications will be more efficient.

A more difficult situation arises if the coarsest grid can not—or should not—be partitioned equitably among the processors. Some of the processors will then not receive any points at lower levels of refinement, but they should receive their fair share of points at the higher levels. Several strategies are conceivable to deal with this problem.

The simplest is to assign all points to a single processor once the grid becomes too coarse (`CHN_Set_solopartition_cuts/owners`). This is quite efficient if there are many levels of refinement, because it reduces the number of communications on coarse grids, while the communication volume is already small. The strategy can be implemented conveniently if two decompositions are defined for the grid at the base level k^b, which is the coarsest level that is partitioned equitably. The first is equitably partitioned, and the second is the solo-partition. Switching between the two is accomplished using `CHN_Redistribute`. Taking this approach, we never need explicit communications to move data between grid levels.

Another strategy is to drop every other cut at each lower level of refinement below the base level. Ownership of cells at the lower level can be derived from that at the higher level by inspecting the eight level-k cells logically contained in each level-$(k-1)$ cell ($k \leq k^b$); for example, we can assign ownership of the level-$(k-1)$ cell to the processor that also owns the $(0,0,0)$-element of the $(2\times 2\times 2)$-cube of cells at level k. This is easy to program, and ascertains that at least some data is already in the right place (owned by the same processor as before). Now it may be advantageous to define two decompositions at *each* level up to and including k^b, and to use redistributions to map between them.

Bibliography

[1] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, S.K. Weeratunga, *The NAS Parallel Benchmarks*, Int. J. Supercomputing Applications, Vol. 5, No. 3, pp. 63-73, 1991

[2] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, M. Yarrow, *The NAS parallel benchmarks 2.0*, Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA 94035, December 1995

[3] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries*, Modern Software Tools in Scientific Computing, E. Arge, A.M. Bruaset, H.P. Langtangen, Ed., Birkhauser Press, 1997

[4] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, *PETSc 2.0 Users manual*, Report ANL–95/11 - Revision 2.0.17, Argonne National Laboratory, Argonne, IL 60439, 1997

[5] D.L. Brown, G.S. Chesshire, W.D. Henshaw, D.J. Quinlan, *Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[6] J. Bruno, P.R. Cappello, *Implementing the Beam and Warming method on the hypercube*, Proc. 3^{rd} Conf. Hypercube Concurrent Computers and Applications, Pasadena, CA, January 1988

[7] N. Carriero, D. Gelernter, *How to write parallel programs: a guide to the perplexed*, ACM Computing Surveys, Vol. 21, No. 3, pp. 323–357, 1989

[8] Culler et al., *Parallel programming in Split-C*, Proc. Supercomputing '93, Portland, OR, pp. 262–273, November 1993

[9] L. Dagum, *OpenMP: A proposed standard API for shared memory programming*, URL: "http://www.openmp.org"

[10] S.J. Fink, S.B. Baden, S.R. Kohn, *Flexible communication mechanisms for dynamic structured applications*, Proc. Third International Workshop on Parallel Algorithms for Irregularly Structured Problems, Santa Barbara, CA, August 1996

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine*, MIT Press, Cambridge, MA, 1994.

[12] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, M.S. Lam, *Maximizing Multiprocessor Performance with the SUIF Compiler*, IEEE Computer, December 1996.

[13] S. Hiranandani, K. Kennedy, C. Tseng, *Preliminary experiences with the Fortran D compiler*, Supercomputing '93, Portland, OR, pp. 338–350, November 1993

[14] E. Johnson, D. Gannon, *Programming with the HPC++ parallel Standard Template Library*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[15] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994

[16] P.J. Kominsky, *Performance analysis of an implementation of the Beam and Warming implicit factored scheme on the NCube hypercube*, Proc. 3^{rd} Symposium Frontiers of Massively Parallel Computation, College Park, MD, October 1990

[17] L. Lamport, *The parallel execution of DO loops*, Communications of the ACM, Vol. 17, No. 2, 1974.

[18] MPI Forum, *MPI-2: Extensions to the Message-Passing Interface*, URL: "http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2-report/mpi2-report.html"

[19] N.H. Naik, V.K. Naik, M. Nicoules, *Parallelization of a class of implicit finite difference schemes in computational fluid dynamics*, Int. J. High Speed Computing, Vol. 5, No. 1, pp. 1–50, 1993

[20] J. Nieplocha, R.J. Harrison, R.J. Littlefield, *The global array programming model for high performance scientific computing*, SIAM News, Vol. 28, No. 7, August-September 1995

[21] Parallel Computing Forum, *Parallel Extensions for Fortran*, X3H5/93-SD1-Revision L, URL: "http://www.cs.orst.edu/standards/ANSI-X3H5/94/"

[22] S.J. Scherr, *Implementation of an explicit Navier-Stokes algorithm on a distributed memory parallel computer* AIAA Paper 93-0063, 31^{st} Aerospace Sciences Meeting and Exhibit, Reno, NV, 1993

[23] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1995.

[24] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, *PARTI primitives for unstructured and block structured problems*, Computing Systems in Engineering (Proc. Noor's Flight Systems Conference), pp. 73-86, Vol. 3, No. 1, 1992

[25] R.F. Van der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, Proc. Supercomputing '93, Portland, OR, November 1993

[26] R.F. Van der Wijngaart, M. Yarrow, M.H. Smith, *An architecture-independent parallel implicit flow solver with efficient I/O*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[27] R.F. Van der Wijngaart, *Charon toolkit for parallel, implicit structured-grid computations: Literature survey and conceptual design*, NAS Report 97-019, NASA Ames Research Center, Moffett Field, CA, 1997

Index

List of constants

User parameters

Boolean return values, user error return codes

CHN_SUCCESS – 0 (no error)
CHN_ERR_ACCESS_TOGGLE – illegal attempt to (re)set ghost access
CHN_ERR_ADDRESS – illegal starting address
CHN_ERR_BUF_SIZE – buffer size not allowed (serial I/O)
CHN_ERR_CELL – incorrect cell number
CHN_ERR_COMM – incorrect communicator
CHN_ERR_CUT – incorrect number of cuts, cut value, or index
CHN_ERR_DATA_TYPE – illegal data type
CHN_ERR_DECOMPOSITION – incorrect decomposition type
CHN_ERR_DIMENSION – incorrect dimension(ality)
CHN_ERR_ERROR_MODE – incorrect error handling mode
CHN_ERR_FILE_NAME – name of file specified for I/O contains only blanks
CHN_ERR_FILE_NAME_LENGTH – file name too long
CHN_ERR_FILE_NOT_NAMED – file specified for I/O has no name
CHN_ERR_FILE_NOT_OPENED – file specified for I/O not opened
CHN_ERR_FILE_TYPE – file specified for I/O not binary direct access
CHN_ERR_GHOST_POINTS – illegal number of ghost points
CHN_ERR_GRID_MISMATCH – incompatible grids in redistribute
CHN_ERR_HANDLE – incorrect handle
CHN_ERR_INDEX – incorrect tensor component
CHN_ERR_KEY – illegal key
CHN_ERR_LOCAL_TOGGLE – illegal attempt to (re)set local execution mode
CHN_ERR_MASK – invalid or incompatible tensor mask
CHN_ERR_MEMORY_TYPE – incorrect type specified for system-claimed memory
CHN_ERR_NONLOCAL_VALUE – processor does not own value(s), but assign mode is local
CHN_ERR_NUM_BUFFERS – illegal number of CHN_Mvalue buffers
CHN_ERR_NUM_SUBSCRIPTS – illegal number of subscripts

CHN_ERR_OWNER – illegal owner
CHN_ERR_PANEL – incorrect panel specification
CHN_ERR_PERIODIC – illegal periodicity parameter
CHN_ERR_POINT – point not in grid
CHN_ERR_RANK – illegal tensor index or rank
CHN_ERR_REMEMBER_TOGGLE – illegal attempt to (re)set communication caching
CHN_ERR_REMEMBERING – cannot delete communication caching key within caching bracket
CHN_ERR_SECTION – cannot place cuts in section
CHN_ERR_SHAPE – illegal shape (when defining unipartition section)
CHN_ERR_SIDE – illegal side (used in copying faces)
CHN_ERR_SIZE – incorrect grid, array, or tensor size
CHN_ERR_TILE – incorrect tile specification

System error return codes

CHN_ERR_ALLGATHERV
CHN_ERR_ALLOC
CHN_ERR_ATTR_GET
CHN_ERR_ATTR_PUT
CHN_ERR_BCAST
CHN_ERR_CREATE_KEYVAL
CHN_ERR_DELETE_KEY
CHN_ERR_GET_TILE
CHN_ERR_HASH_INSERT
CHN_ERR_INIT_TABLE
CHN_ERR_IRECV
CHN_ERR_MASK_ANALYSIS
CHN_ERR_NEW_HANDLE
CHN_ERR_NOT_COMMITTED
CHN_ERR_PUT_TILE
CHN_ERR_READ
CHN_ERR_REDUCE
CHN_ERR_RESET_BUFS
CHN_ERR_SEND
CHN_ERR_SORT_LIST
CHN_ERR_SPEED_EXECUTE
CHN_ERR_STORE_ALLGATHERV
CHN_ERR_STORE_BCAST
CHN_ERR_STORE_DRAIN_BUF
CHN_ERR_STORE_FILL_BUF
CHN_ERR_STORE_IRECV
CHN_ERR_STORE_REDUCE
CHN_ERR_STORE_RESET_BUFS
CHN_ERR_STORE_SEND
CHN_ERR_STORE_TEMP_REQUEST
CHN_ERR_STORE_TEMP_SIZE
CHN_ERR_STORE_WAIT
CHN_ERR_WAIT
CHN_ERR_WRITE

Library functions

p: procedure; *f*: function returning value